Towards Practical Oblivious Map

Xinle Cao[*]
Zhejiang University
xinle@zju.edu.cn

Weiqi Feng[*]
University of Massachusetts Amherst
weiqifeng@umass.edu

Jian Liu
Zhejiang University
liujian2411@zju.edu.cn

Jinjin Zhou
Ant Group
zhoujinjin.zjj@antgroup.com

Wenjing Fang
Ant Group
bean.fwj@antgroup.com

Lei Wang
Ant Group
shensi.wl@antgroup.com

Quanqing Xu
OceanBase, Ant Group
xuquanqing.xqq@oceanbase.com

Chuanhui Yang
OceanBase, Ant Group
rizhao.ych@oceanbase.com

Kui Ren
Zhejiang University
kuiren@zju.edu.cn

ABSTRACT

Oblivious map (OMAP) is an important component in encrypted databases, utilized to safeguard against the server inferring sensitive information about client's encrypted key-value stores based on *access patterns*. Despite its widespread usage and importance, existing OMAP solutions face practical challenges, including the need for a large number of interaction rounds between the client and server, as well as the substantial communication bandwidth requirements. For example, the state-of-the-art protocol named OMIX++ in VLDB 2024 still requires $O(\log n)$ interaction rounds and $O(\log^2 n)$ communication bandwidth per access, where n denotes the total number of key-value pairs stored.

In this work, we introduce more practical and efficient OMAP constructions. Consistent with all prior OMAPs, our constructions also adapt only the *tree-based Oblivious RAM (ORAM)* and *oblivious data structures* (ODS) to achieve OMAP for enhanced practicality. In complexity, our approach needs $O(\log n/\log\log n) + O(\log\lambda)$ interaction rounds and $O(\log^2 n/\log\log n) + O(\log\lambda\log n)$ communication bandwidth per data access where λ is the security parameter. This new complexity results from our two main contributions. First, unlike prior works that rely solely on *search trees*, we design a novel framework for OMAP that combines *hash table* with search trees. Second, we propose a more efficient tree-based ORAM named DAORAM, which is of significant independent interest. This newly developed ORAM noticeably accelerates our constructions as it supports obliviously accessing hash tables much more efficiently. We implement both our proposed constructions and prior methods to experimentally demonstrate that our constructions substantially outperform prior methods in terms of efficiency.

1 INTRODUCTION

Oblivious algorithms [10, 16, 38, 70, 79] serve as a critical mechanism frequently employed alongside encrypted databases (EDBs) [23, 30, 59] to uphold users' data privacy. They ensure that the access patterns remain independent of the database contents [37]. Therefore, during query processing, an untrusted server gains no information beyond query types, database size, and the size of query results [15]. Recently, there has been a surge in the usage of oblivious algorithms with EDBs [20, 44, 53, 83], which is driven by concerns regarding the security implications of query access pattern leakages [51, 55, 62].

The *oblivious map* (OMAP) [67, 79] is a specific type of oblivious algorithm designed to facilitate oblivious access to key-value (KV) stores [67], one of the most used database formats in real-world applications [29, 42]. OMAP offers clients the security guarantee that an untrusted server, holding the encrypted KV pairs, cannot obtain information regarding which data pair was accessed during query processing, nor its content. Furthermore, OMAP is often used to construct oblivious algorithms for executing more complex queries such as join [20], aggregate [30], and range query [19] in other types of databases. However, designing efficient and practical OMAPs presents a significant challenge. Predominantly, most existing oblivious algorithms rely on the established cryptographic primitive called oblivious RAM (ORAM) [37]. This primitive is a generic tool for achieving obliviousness as it was originally proposed to access *memory* obliviously in random access machine. Specifically, given KV pairs $\{(k_i, v_i)\}_{i=0}^{n-1}$ where keys are consecutive integers (which is used to simulate memory), ORAM supports obliviously accessing one pair from them in functionality. While sharing similarities with ORAM, OMAP is more general and powerful since OMAP supports KV stores, even when the keys are **non-consecutive and arbitrary strings**. This difference incurs a huge gap in their designs such that OMAP cannot be naively constructed from ORAMs with practicality. It raises the following important question and motivated a series of works [15, 30, 67, 79]:

How can we design an efficient OMAP based on practical ORAMs, requiring only small client-side storage like $O(\log n)$?

Oblivious data structure. Some prior works [15, 30, 67, 79] also attempt to address OMAP through another way, i.e., the use of oblivious data structure (ODS). In short, ODS refers to oblivious algorithms designed specially for some data structures such as trees and stacks in order to support obliviously accessing these structures *more efficiently* than using the generic ORAM. Wang et al. [79] are the first to define ODS and non-trivially adapt tree-based ORAM to achieve this goal. They introduced an OMAP construction employing ODS for an AVL tree, ensuring that client-side storage does not exceed $O(\log n)$. Since then, this construction has been widely implemented in plenty of works [12, 28, 36] due to its simplicity. The state-of-the-art work on OMAP [15, 75] continues to use this approach as a foundation, incorporating several new optimizations. While the AVL tree makes the OMAP have a good theoretical communication bandwidth, it may not be the optimal choice among

[*]These authors contributed equally to this work.

ODS Method	Interaction Round	Communication Bandwidth	Note
ODS+AVL [15, 79]	$O(\log n)$	$O(\log^2 n)$	Many interaction rounds
ODS+B/B+ [20, 30, 67]	$O(\log n/\log \beta)$	$O(\beta \log^2 n/\log \beta)$	Larger bandwidth blowup
Ours	$O(\log n/\log\log n) + O(\log \lambda)$	$O(\log^2 n/\log\log n) + O(\log n \log \lambda)$	Better rounds and bandwidth in practice

Table 1: Comparison of approaches for oblivious map. n is the number of KV pairs stored, λ is the security parameter. β is a *constant* set for branching factor in B/B+ tree [30, 67], thus ObliDB [30] still expresses its interaction and bandwidth as $O(\log n)$ and $O(\log^2 n)$, respectively.

search trees in practice, as noted in [67, 79]. For instance, it requires $3 \cdot 1.44 \log n$ interaction rounds.

Consequently, some works introduce OMAPs based on other types of search trees, including B/B+ trees [20, 30] and a variant similar to B-trees [67], to reduce interaction rounds and improve efficiency. However, these new OMAPs reduce interaction rounds at the expense of increased theoretical communication bandwidth. We summarize the complexity of all existing OMAPs that adapt only the practical tree-based ORAMs in Table 1. The table demonstrates that, to achieve $O(\log n/\log \beta)$ interaction rounds per access, prior works require a larger communication bandwidth of $O(\log^2 n/\log \beta)$, where β is a constant integer predefined by the client. The value of β implies a trade-off between interaction rounds and communication bandwidth. It cannot be too large, as this would result in prohibitively high bandwidth costs. For instance, with $\beta = n$, the communication bandwidth reaches $O(n)$, equivalent to downloading the entire database. Therefore, the value of β must be chosen carefully to adapt to specific applications. Additionally, the communication bandwidth remains still $O(\log^2 n)$ regardless of β, which can be a bottleneck for OMAP when implemented in secure enclaves [75]. Therefore, we ask the following question:

Can we propose new ODS that achieve both fewer interaction rounds and reduced communication bandwidth for more efficient OMAPs?

In this work, we revisit the two questions above and provide a positive answer. Specifically, we propose several new constructions which **are the first to build OMAPs via combining both new novel tree-based ORAMs and ODSs**. These constructions are the first to overcome the $O(\log^2 n)$ communication bandwidth barrier, marking a significant theoretical improvement in OMAP bandwidth [75]. Furthermore, they require only $O(\log n/\log\log n)$ interaction rounds per operation. Based on these merits, the proposed methods are far more efficient than prior approaches.

1.1 Overview

Framework. We first introduce a new simple but effective framework for designing more efficient OMAPs. Prior methods organize KV pairs as a search tree and then construct an ODS for the search tree. To improve this approach, we explore the use of hash tables, which are well-known for their efficiency in mapping [24]. However, oblivious hash tables are not ideal in this context due to their expensive costs for achieving obliviousness. As discussed in [79], oblivious hash tales can be achieved via a tree-based ORAM [73] with only $O(\log n)$ client-side storage, but one access to the table requires three accesses to the ORAM for addressing collision in the table. As accessing the tree-based ORAMs with the $O(\log n)$ client-side storage is often costly, i.e., $O(\log n)$ interaction rounds and $O(\log^2 n)$ communication bandwidth per access, oblivious hash tables via ORAMs are considered to have the same complexity and impracticality as OMAPs constructed from ODS for the search tree.

Surprisingly, recent works over the last decade [32, 47] demonstrate that accessing the tree-based ORAMs with limited client-side storage can be done more efficiently. Nevertheless, they still encounter some non-trivial practical problems, which prevent their easy adaption in real-world implementations. In this paper, we will show *how to address these problems* elegantly and propose a much more practical and efficient ORAM protocol called DAORAM (**d**e-**a**rmotized ORAM), which is a contribution of substantial independent interest. DAORAM can complete each access with only $O(\log n/\log\log n)$ interaction rounds and $O(\log^2 n/\log\log n)$ bandwidth. With the new advanced ORAM, now we can follow the approach in [79] to naively achieve an oblivious hash table without collision and an OMAP with better complexity. To obtain even more optimized OMAP constructions, we propose a new framework consisting of two components, as outlined below:

- **ORAM for hash table.** We initialize an ORAM to store a hash table of size n *which allows collisions*. For each integer $i \in \{0, \dots, n-1\}$, we map i to g_i and store this mapping in the ORAM, where g_i is used to record the group of collided KV pairs, i.e., any (k, v) such that $\mathsf{Hash}(k) = i$, where $\mathsf{Hash}(\cdot)$ is a hash function randomly mapping a string to an integer in $\{0, \dots, n-1\}$.
- **Group OMAP.** The length of g_i is limited and cannot store all KV pairs mapped to i. To address this, we establish an OMAP for all collided pairs at position i. Thus, g_i only needs to store metadata about the OMAP, requiring only $O(\log n)$ bits. There are n distinct OMAPs, as we build an OMAP for each g_i where $i \in \{0, \dots, n-1\}$. We store them in the same ORAM to prevent the server from observing *which group OMAP* is accessed during query processing.

To summarize, *we handle the collisions in hash tables by utilizing smaller OMAPs for collided KV pairs*. When the client accesses a KV pair (k, v) where $\mathsf{Hash}(k) = j$, it retrieves the corresponding g_j from the hash table ORAM. Then, it uses g_j to find (k, v) from the OMAP storing the collided pairs. The overhead of our OMAP is equal to the sum of the overhead in accessing the ORAM and the group OMAP. Importantly, accessing the group OMAP can be much more efficient than accessing the ORAM as each group has at most $O(\lambda)$ collided pairs [27, 65]. Hence, our framework allows us to use *only one access to the ORAM and a much cheaper access to the group OMAP* to replace the *three accesses* to the ORAM in the general construction of an oblivious hash table without collision proposed by [79], significantly improving practicality.

Contributions. We summarize our contributions below.

(1) *A new OMAP framework.* We propose a new OMAP framework that combines both ORAM for hash tables and ODS for search trees. Within this framework, we can apply a prior theoretically elegant *tree-based* ORAM scheme [47] and existing OMAPs[15, 30, 67, 79] to present several new OMAP constructions. Compared with prior OMAPs, they are **asymptotically better** and *do not* require any additional expensive techniques.

(2) *A faster ORAM.* We identify the infeasible worst-case performance and impracticality of the ORAM protocol [47] used in our constructions, which makes them unacceptable in production. To this end, we introduce a new **d**e-**a**rmotized ORAM protocol named DAORAM. It offers substantially better performance and greater practicality compared to [47], making our constructions indeed outperform all prior OMAPs **both theoretically and practically**.
(3) *Full-fledged Implementation.* We implement three typical prior OMAPs including the widely used baseline [79] and SOTA works [15, 30] and our three new OMAP constructions based on them. We provide a comprehensive evaluation of our DAORAM and OMAPs, demonstrating the significant speedup of our framework to prior OMAPs. The experimental results show that our OMAPs improve **processing time by up to 71.4% and communication bandwidth by up to 74.1%** compared to the SOTA work [15].

2 PRELIMINARIES

In this section, we introduce some basic and important notions used in this work. All notations in this work are introduced as needed, and all algorithms including the adversary are assumed to be probabilistic polynomial-time (PPT).

Pseudorandom function. Following [50], we call a function $F : \{0,1\}^{k_1} \times \{0,1\}^{k_2} \to \{0,1\}^{k_3}$ a *pseudorandom function* (PRF) if:

- There is a polynomial-time algorithm: given a key $K \in \{0,1\}^{k_1}$ and an input $x \in \{0,1\}^{k_2}$, it computes $F_K(x) = F(K, x)$.
- For any PPT adversary $\mathcal{A}$, its advantage

$$\mathsf{Adv}_F^{\mathsf{prf}}(\mathcal{A}) = |\Pr_{K \leftarrow\$ \{0,1\}^{k_1}}[\mathcal{A}^{F_K(\cdot)} = 1] - \Pr[\mathcal{A}^{\$} = 1]|$$

  is negligible in λ, where \$ above denotes the oracle that implements a random function from $\{0,1\}^{k_2}$ to $\{0,1\}^{k_3}$, $\mathcal{A}^{F_K(\cdot)}$ and $\mathcal{A}^{\$}$ denote that the adversary has access to the oracle of function $F_K(\cdot)$ and random function, respectively.

ORAM and OMAP. Oblivious RAM (ORAM) and oblivious map (OMAP) are very similar in both definition and functionality. Generally speaking, both of them allow the client C to store a database $\mathcal{DB} := \{(k_i, v_i)\}_{i=1}^n$ encrypted on the untrusted server $\mathcal{S}$ and then operate each pair of data *obliviously*, i.e., $\mathcal{S}$ cannot infer which pair is operated by C via observing access patterns during operations (refer to Section 4.1 for the formal definition). However, they are very different in key-value (KV) stores supported: ORAM is originally proposed to access **memory** obliviously. So it always assumes keys in KV store are consecutive integers to simulate memory. OMAP is more general and powerful as it is designed for all KV stores where keys can be arbitrary and non-consecutive strings. To this end, there is a huge gap between existing ORAMs and OMAPs expected in deployment:

- **Feasible but impractical:** Actually, there are indeed some existing ORAMs (e.g., hierarchy ORAMs [5, 63, 64]) which naturally support non-consecutive keys. Nevertheless, they are highly inefficient due to the large constant factors in overhead complexity, even though some of them [5, 7] achieve the $O(\log n)$ optimal theoretical communication bandwidth of ORAM [56].

Notation	Description
C	the client
$\mathcal{S}$	the untrusted server
$\mathcal{DB}$	KV Database of client
n	number of KV pairs in $\mathcal{DB}$
m	number of operations to $\mathcal{DB}$
$\{(k_i, v_i)\}_{i=1}^n$	KV pairs with arbitrary keys
$\{(op_i, ek_i, ev_i)\}_{i=1}^m$	operation sequence on $\mathcal{DB}$
$[N]$	consecutive integer set $\{0, ..., N-1\}$
$x \leftarrow\$ I$	uniformly sampling x from I
λ	security parameter
pt	the path label for the KV pair
$\mathcal{D}$	distribution
$\mathcal{G}_i$	the group of collided pairs mapped to i
X	the recursion degree
GC	the global counter
IC	the individual counter
α	the length of GC
γ	the length of IC
sk	secret key for encryption
K	secret key for PRF
$\equiv$	statistically indistinguishable
$\equiv_c$	computationally indistinguishable

Table 2: Summary of notations.

- **Practical but infeasible:** When we try more practical ORAMs, only tree-based ORAMs [73, 78] demonstrate relative efficiency and are widely used in EDBs [11, 12, 20, 81]. But tree-based ORAMs, when constrained by *limited client-side storage*, typically $O(\log n)$ for practical applications, are capable of supporting only *consecutive* keys as the ORAM functionality requires. They cannot be naively applied to process a KV store with unpredictable and non-consecutive keys, e.g., the database $\mathcal{DB} := \{((\mathsf{Alice}, \mathsf{Boston}), (\mathsf{Bob}, \mathsf{London}), \cdots)\}$.

These limitations above leave building practical OMAPs via ORAM still unsolved. As OMAP is a fundamental primitive for oblivious algorithms and secure EDBs, building practical OMAPs becomes an imperative task in encrypted databases (EDBs).

In algorithms, both ORAM and OMAP consist of two subroutines:

- **Initialization:** $\mathsf{Init}(n, \lambda) \to (\mathsf{st}_C, \mathsf{st}_{\mathcal{S}})$. On input the (estimated) maximal number of pairs in the database n and security parameter λ. C and $\mathcal{S}$ interact with each other to run this subroutine, and produce client state st_C in C and server state $\mathsf{st}_{\mathcal{S}}$ in $\mathcal{S}$.
- **Access:** $\mathsf{Access}(\mathsf{st}_C, \mathsf{st}_{\mathcal{S}}, k, v) \to (\mathsf{st}'_C, \mathsf{st}'_{\mathcal{S}}, v')$. On input the states $(\mathsf{st}_C, \mathsf{st}_{\mathcal{S}})$ and a pair (k, v), C and $\mathcal{S}$ interact with each other to run this subroutine, and produce the updated states $(\mathsf{st}'_C, \mathsf{st}'_{\mathcal{S}})$. If v is $\perp$, then this is an read operation; v' is set to the value stored in $\mathsf{st}_{\mathcal{S}}$ corresponding to k. Otherwise, this is a write operation, and (k, v) is written in $\mathsf{st}'_{\mathcal{S}}$, where $v' = v$.

ODS. Differing from the ORAM and OMAP primitives that aim to operate a single KV pair, oblivious data structure (ODS) [79] tries

to design oblivious algorithms specialized to some data structures, e.g., trees [67], heaps [71], and graphs [15]. This enables operating these data structure more efficiently than using the generic ORAM/OMAP [48]. In other words, while ORAM/OMAP is a generic primitive to build various oblivious algorithms, ODS is the specialized data structure for optimizing some important oblivious algorithms in applications, e.g., ObliDB [30] builds oblivious B+ tree to complete range queries obliviously instead of ORAM/OMAP. Additionally, we remark that, although OMAP is conceptually similar to ORAM, compared with extending ORAM to achieve OMAP, most existing works often achieve OMAP via the ODS for search trees [15, 30, 79] to enhance practicality.

3 REVISIT

In this section, we introduce some intuition and specific constructions of prior ORAM/OMAP. They are necessary components this work is based on. Especially, we revisit them to point out their shortages, explaining why more advanced constructions are needed.

3.1 Prior Recursive ORAM

3.1.1 Basic Intuition. We first provide some background on how tree-based ORAMs work. As illustrated in Figure 1, a KV pair (k, v) is assigned a random *label*, denoted by $pt \in [n]$, indicating the path this pair is stored on. $\mathcal{S}$ stores a tree where the ciphertext of (k, v, pt) is guaranteed to be on the path from the root node to the pt-th leaf node. The position map in $\mathcal{C}$ records the corresponding pt for each key, resulting in a size of $O(n)$. Each time $\mathcal{C}$ searches for (k, v), it first retrieves the corresponding pt from the position map using k and then accesses the path indicated by pt (the green path in Figure 1). After $\mathcal{C}$ retrieves the pair from the path, its label will be replaced by a new random value, denoted by pt'. This means the pair will be placed in the path of pt'. To achieve this, $\mathcal{C}$ can adopt different eviction strategies [21, 73, 78] to balance various trade-offs. If the eviction process fails, the pair is temporarily placed in the stash and will be retried to evict during the next ORAM access. It has been proven that with some existing strategies [73, 78], the stash size exceeds $O(\log n)$ with a negligible probability.

While tree-based ORAM looks perfect, the $O(n)$ position map storage actually makes tree-based ORAM impractical. To address this, prior works [72, 74] introduce a technique named *recursion*: it uses a series of smaller ORAMs to store the position map but requires the **keys must be consecutive integers**. For example, suppose the position map records $\{(ck_i, pt_i)\}_{i=0}^{n-1}$ where ck denotes consecutive integers, then $\mathcal{C}$ can use $\lceil n/2 \rceil$ blocks in another ORAM to store them, where the ith block records $\{(ck_{2i}, pt_{2i}), (ck_{2i+1}, pt_{2i+1})\}$. Such an ORAM and block are called PosMap ORAM and block, respectively. To distinguish them from the original ORAM and block holding KV pairs, we call the original ORAM and block as data ORAM and block, respectively. Besides the number 2, the recursion can be deeper with a larger number here. We call this number recursion degree and denote it as X.

In the most common setting [73], the size of both data blocks and PosMap blocks are set to be $O(\log n)$ [32, 73], the path label pt also needs $\log n$ bits to record the corresponding path. In this way, X can be only a constant. The above example shows that we can apply a PosMap ORAM with $\lceil n/X \rceil$ blocks to store the position map for the data ORAM. As X is a constant, this *recursion* process needs to be repeated for $O(\log n)$ times such that $\mathcal{C}$ can ultimately use constant storage to access the data ORAM. Unfortunately, the recursion process requires $\mathcal{C}$ to sequentially access PosMap ORAMs from small to large and finally access the data ORAM, incurring $O(\log n)$ interaction rounds and $O(\log^2 n)$ communication bandwidth between $\mathcal{C}$ and $\mathcal{S}$. Such expensive costs makes recursive ORAM impractical and motivate some works [32, 47] to improve the recursion process.

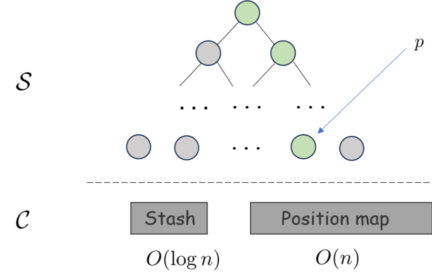


Figure 1: The illustration of tree-based ORAMs.

3.1.2 Review. Here we review some works that try to enlarge X to be $O(\log n / \log\log n)$ in recursive ORAMs to enhance practicality. Fletcher et al. are the first to enlarge X but in an insecure way, which was fixed by Chan et al. [47] later. For ease of understanding, throughout this paper, we treat the recursion process as traversing a complete X-ary tree and here call each pair as a node in the tree. A KV pair in the PosMap ORAM preserving the index (i.e., pt) of X pairs in the next larger ORAM is described as one internal node recording the index of its X children.

The main idea of Fletcher et al. [32] is using PRF to generate the index instead of recording the index. In detail, each internal node in [32] consists of three parts: (1) a $\log n$-bit key and $\log n$-bit path; (2) a α-bit group counter (GC); (3) X γ-bit individual counters (ICs):

$$\mathsf{id}||pt||\mathsf{GC}||\mathsf{IC}_0||\cdots||\mathsf{IC}_{X-1}.$$

where id is the key of this node and pt is the index (the path where this node is within the ORAM). The values of GC and ICs are initialized as 0. To keep the $O(\log n)$ node size, it is required that $\alpha + \gamma \cdot X \sim O(\log n)$. For this internal node, the recursion process guarantees that the keys of its children are $\{a, a+1, ..., a+X-1\}$ where $a = X \cdot \mathsf{id}$. That's why this node does *not need to store these keys*, leaving the potential to enlarge X. To determine the path of the child with key $a+j$ ($j \in [X]$), $\mathcal{C}$ calculates this path based on PRF function, GC, and IC_j. Specifically, $\mathcal{C}$ maintains a secret key K for a PRF function PRF and generates:

$$pt_{a+j} := \mathsf{PRF}_{\mathsf{sk}}(a + j \| \mathsf{GC} \| \mathsf{IC}_j)^1. \quad (1)$$

In the Initialization procedure, $\mathcal{C}$ assigns pt_{a+j} to the child with key $a+j$ as its index and this child will be guaranteed to be in the path corresponding to pt_{a+j}. For Access procedure, when $\mathcal{C}$ wants to retrieve this child, it gets GC and IC_{a+j} during recursion, calculates pt_{a+j}, and retrieves this path to get this child. After the retrieval, $\mathcal{C}$ executes increment:

$$\mathsf{IC}_j := \mathsf{IC}_j + 1 \mod 2^\gamma$$

and reassigns a new path to this child with Equation 1 for eviction placing this child back to the ORAM. In this way, the length of IC can

[1] The level of current node is also taken as an input of PRF, we follow [32] to omit it throughout this paper for ease of presentation.

be $o(\log n)$ to allow a larger X. For example, setting $\gamma \sim O(\log\log n)$ and $\alpha \sim O(\log n)$, then they enable $X \sim O(\log n/\log\log n)$.

Security and Fix. There is a vulnerability in the original construction of Fletcher et al. [32]: the value of $\mathsf{GC}||\mathsf{IC}_j$ should not be repeated for any $j \in [X]$ for satisfying computational security. So after C accesses the node with index $a + j$ for $2^\gamma - 1$ times, i.e., the value of IC_j is going to be repeated in the next access towards this node, C is required to change the value of GC and update the path of all the X nodes with the updated GC. This process is called *reset*:

(1) Before updating GC, C retrieves all the nodes with key $\{a, a+1, ..., a+X-1\}$ according to GC and ICs.
(2) C updates $\mathsf{GC} := \mathsf{GC} + 1$, then sets $\forall j \in [X], \mathsf{IC}_j := 0$. Finally, C assigns each node with the new path calculated based on the updated GC and ICs and places them back using eviction.

In the above process, C is required to retrieve and return all X nodes above, making it much expensive. Worse more, the reset happens only when one IC is going to be repeated. As pointed out by Chan et al. [47], now the adversary can infer sensitive information according to the reset frequency. For example, if C is always accessing the same node, then the reset happens very frequently because the same IC is always incremented. However, if C accesses all distinct nodes, no IC is repeated, C will never do reset. So the adversary can infer the pair access distributions according to the reset frequency.

Chan et al. [47] propose a *theoretically* elegant fix where the reset is done randomly. In each access to a child, they do the reset with a probability of $1/X$. So the reset is done independent of the access distribution. In this case, Chan et al. need to guarantee that before any IC is repeated, the reset must have been done to this node. Therefore, they require $\gamma = 3\log\log n$ when X is $\log n/\log\log n$. This promises that repeated $\mathsf{GC}||\mathsf{IC}$ happens with a probability of $(1-1/X)^{2^\gamma}$ which is negligible in n [47]. Now the cost of reset is still $O(X \log n)$ and the reset is expected to happen once every X accesses. So under this fixed solution, the interaction round and communication bandwidth are $O(\log n/\log X)$ and $O(\log^2 n/\log\log X)$, respectively.

3.1.3 *Observations.*

The fixed approach by Chan et al. [47] is theoretically elegant but leaves some drawbacks in practicality. Here we point out these shortages and *we will address all of them with our new construction in Section 5.*

Observation 1 (amortized). *The fixed approach guarantees* **only amortized** *interaction rounds of $O(\log n/\log\log n)$ and communication bandwidth of $O(\log^2 n/\log\log n)$.*

This observation is due to the probabilistic reset operations. Suppose the client can store and retrieve at most μ (μ should be a constant) paths once, then the interaction rounds per query are

$$\left\lceil \frac{\log n}{\log X} \right\rceil + \frac{X \cdot u}{\mu}$$

where u is the number of reset operations triggered in the query processing. Also, the communication bandwidth is

$$\log n \cdot \frac{\log n}{\log X} + X \cdot u \cdot \log n.$$

Note u follows the binomial distribution, i.e., $u \sim \mathsf{Bin}(\lceil \frac{\log n}{\log X} \rceil, \frac{1}{X})$. So the query performance actually fluctuates, in the worst case where $u = \lceil \frac{\log n}{\log X} \rceil$, if we assume C can only store one path in local once, the interaction rounds required are as $(X+1)$ times as that in the best case where $u = 0$. And obviously, the theoretical complexity in the worst case is also much larger than the amortized complexity. To this end, it is essential to study if we can do deamortization [7, 17, 54, 61] here, i.e., improving the worst-case performance while preserving efficiency. Note *performing stably* is an important property in production [39] and all prior OMAPs and tree-based ORAMs satisfy it, thus without this property, ORAMs and OMAPs may be not competitive to prior works.

Observation 2 (Strict Parameters). *The fix requires γ must be no smaller than* $3\log\log n$ *and n to be large to guarantee negligible probability* $(1-1/X)^{2^\gamma}$.

These strict parameter values affect the actual performance of the fixed solution. The value of γ implies that if the block size is fixed (like memory blocks), then the upper bound of X is also fixed because we cannot change γ to smaller values than $3\log\log n$. However, with a small X, there can be still too many expensive interaction rounds, making the ORAM inefficient. So we wonder if the value of γ can be smaller for better efficiency. While the smaller values do not imply the improvement on complexity, they are important for actual performance. Besides, another important issue is if we can achieve the security on resets **perfectly**: whatever n is, it is guaranteed that reset must happen before $\mathsf{GC}||\mathsf{IC}$ in a block is repeated without the sacrifice of obliviousness and efficiency.

3.2 Prior OMAPs

The prior OMAPs [15, 30, 67, 79] organize KV pairs as a search tree according to key orders. To access a pair, C traverses the search tree to find it. Typically, these works apply some classic *data-dependent* search tree such as an AVL tree or a B+ tree. These structures are determined by both database sizes and contents. Traversing the tree requires $O(\log_\beta n)$ interaction rounds where β is the branching degree of a node. Achieving OMAPs naturally involves enabling C to traverse the search tree obliviously, which can be done with a *pointer-based technique* [79] in ODS. In Figure 2, we provide a minimal AVL tree example with data pairs $\{(\mathsf{Alice}, \mathsf{Boston}), (\mathsf{Bob}, \mathsf{London}), (\mathsf{Carol}, \mathsf{Paris})\}$. To preserve the node of an AVL tree, the ORAM tree stores 1) the keys of this node and its children and 2) the paths this node and its children are in. Each time it traverses a node, it finds paths its children located on. Finally, C stores only the root node of the AVL tree.

The ODS of data-dependent search trees can achieve the OMAP with only $O(\log n)$ client-side storage where n is the number of KV pairs in the database. However, they incur in-compressible blocks because the client cannot predict the keys and paths of its children, necessitating these values to be recorded in the block. Recall that the block size in ORAMs is assumed to be $O(\log n)$, given that the key length is also at least $O(\log n)$ [79], each block can store only a constant number of keys, i.e., the branching factor of a node in the search tree can be only a constant. In other words, the design of prior works inherently implies the expensive $O(\log n)$ interaction rounds where the complexity constant factor is determined by the block size and n in production. So up to now, the prior OMAPs [15, 30, 67, 79] cannot overcome *either the $O(\log^2 n)$ communication bandwidth or the $O(\log n)$ interaction rounds while not exceeding $O(\log^2 n)$ communication bandwidth.* To this end, this work tries to design

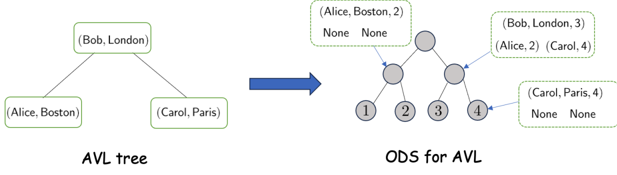


Figure 2: The ODS for AVL. For an AVL node (Bob, London), the ODS block stores not only the KV pair and corresponding path (Bob, London, 3) but also the children keys and paths (Alice, 2) and (Carol, 4).

OMAPs under a new novel framework escaping from the above inherent shortages of data-dependent search trees.

4 OMAP FRAMEWORK

In this section, we define the security model for ORAM/OMAP, then we propose our new novel framework for designing OMAPs. This framework allows us to combine the recursive ORAM introduced in Section 3.1 and (modified) ODS for search trees to instantiate new OMAPs which are *asymptotically better* than prior OMAPs.

4.1 Security Model

Consistent with most ORAMs [63, 73, 78] and EDBs [12, 27, 30], we consider a client C that stores its encrypted database (EDB) on a remote, untrusted server $\mathcal{S}$. Typically, C is assumed to have limited storage [32, 47, 73, 79] to accommodate most devices, including those with very limited resources, such as mobile phones, smartwatches, and secure enclaves [75]. The adversary $\mathcal{A}$ is assumed to be *honest-but-curious* adversary $\mathcal{A}$ to capture $\mathcal{S}$. This adversary does not deviate from the predefined protocols or invade the client C, but it observes everything available on $\mathcal{S}$ in the entire process. Specifically, while C issues read and write operations, $\mathcal{A}$ continuously observes the server state to glean as much sensitive information about C as possible. Below, we provide the formal security notion of ORAM/OMAP:

Definition 4.1 (Security definition). Let $\vec{y}_0 := \{(\mathsf{op}_i, ek_i^0, ev_i^0)\}_{i=0}^{m-1}$ and $\vec{y}_1 := \{(\mathsf{op}_i, ek_i^1, ev_i^1)\}_{i=0}^{m-1}$ denote two operation sequences with the same length m. The operation type op is either read or write. Let $A(\vec{y}_i)$ denote the access sequence of blocks in $\mathcal{S}$ by executing $\vec{y}_i$ via the Access interface of ORAM/OMAP after Initialization[2].

Then an ORAM/OMAP is secure if (1) $A(\vec{y}_0)$ and $A(\vec{y}_1)$ are computationally indistinguishable by $\mathcal{A}$, i.e., they can be distinguished with an advantage of $\mathsf{negl}(\lambda)$ where λ is the security parameter, and (2) it is *correct*, i.e., the results returned by executing $\vec{y}_i$ via ORAM/OMAP is consistent with that returned by executing $\vec{y}_i$ on unencrypted database directly with a probability of $1 - \mathsf{negl}(\lambda)$, which implies the ORAM/OMAP may fail with probability $\mathsf{negl}(\lambda)$.

The definition of OMAP and ORAM differs only in the KV pairs allowed. While OMAP can process operation sequences with arbitrary keys in the KV store, ORAM assumes all the keys in the KV store can be included by an integer interval with the length set by the initialization (which is because ORAM is simulating the memory). The security definition guarantees that the access patterns do not leak information about the operations besides the operation length. The adversary cannot obtain any knowledge about the operation type or content. Similar to prior works [12, 20, 73], we consider the leakage from side-channel attacks, such as when or how frequently C issues requests, to be out of the scope of this paper. More details and effective defenses to these attacks can be found in [22, 33, 43]. While the initial OMAPs treat both search and insertion as *write* operations and hence, indistinguishable. The recent work [15] allows them to be distinguishable for better search efficiency. Our OMAPs can adaptively allow or disallow these two operations to be distinguished as needed. We discuss this and provide formal security proofs of our OMAPs in Appendix C.

[2]The initialization uses the same value for parameter n which is no smaller than m.

4.2 OMAP Framework

In this section, we propose a new framework for designing OMAPs. Compared with prior OMAPs [15, 30, 67, 79], this framework is the first to combine both tree-based ORAMs and ODS of search trees to achieve OMAP. Moreover, using this framework, we propose new OMAPs with the *best-known* complexity on interaction rounds and communication bandwidths under tree-based structures.

Framework. For better efficiency, we redesign the framework to construct OMAP. Specifically, instead of organizing all the KV pairs as a search tree, we follow the design of the hash table in computer science [24] and recent ORAM works [6, 63]. We divide the data pairs into different groups via a hash function and then adopt efficient methods to access each group and the required pair within it, obliviously. In detail, given a database $\mathcal{DB} := \{k_i, v_i\}_{i=0}^{n-1}$, we perform the following steps:

(1) **Hash:** We randomly map each pair to a group with a hash function $\mathsf{Hash} : \{0,1\}^* \rightarrow [n]$:
$$\forall i \in [n], (k_i, v_i) \in \mathcal{G}_j \text{ where } j = \mathsf{Hash}(k_i).$$
It is guaranteed that there are at most $O(\lambda)$ pairs mapped to the same group [35, 72].
(2) **ORAM for consecutive keys:** We apply an ORAM to help access these groups, i.e., we prepare KV pairs $\{(i, pt_i)\}_{i=0}^{n-1}$ where pt_i implies the path to access group $\mathcal{G}_i$. Remark the keys are consecutive so the prior recursive ORAMs [32, 47, 73] (cf. Section 3.1) can be applied here.
(3) **Smaller OMAP for groups:** For each group $\mathcal{G}_i$, we organize pairs within it as a search tree. Then we construct an ODS for the n search trees such that we can access one of them while avoiding C to know which tree is accessed. We can this ODS as *group OMAPs* and it can be achieved by modifying the existing OMAPs [15, 30, 79]. As each group has at most $O(\lambda)$ pairs [27, 65], the interaction rounds are reduced to $O(\log \lambda)$. The ORAM in step (2) uses pt_i to record how to access the search tree here for $\mathcal{G}_i$ and thus we can use it to access any pair in $\mathcal{G}_i$.

To search a pair, C first calculates its group $\mathcal{G}_j$ via the hash function, then obliviously accesses $\mathcal{G}_j$ with ORAM to get pt_j, and finally uses p_j to find the pair via the group OMAP for $\mathcal{G}_j$. Under this new framework, we can apply any ORAM in step (2) and any existing OMAP in step (3) to instantiate OMAP construction. Now we explain how this framework enables more practical OMAPs.

Data-independent Tree. We first discuss the ORAM for consecutive keys, especially the recursive ORAMs introduced in Section 3.1. The recursive access in prior ORAMs can be regarded as obliviously traversing a complete X-ary tree from root to leaf where X is the recursion degree, i.e., **the recursive ORAMs actually establish**

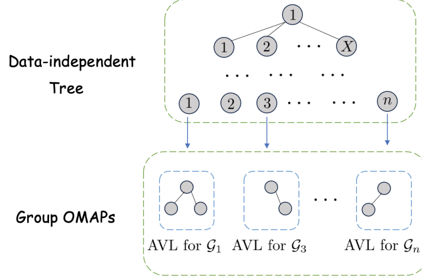


Figure 3: Our constructions under the new framework.

the ODS for a complete X-ary tree. Interestingly, the complete tree is a *data-independent* tree [31], meaning its structure depends on only the database size n. So C can exactly predict the next child node accessed when traversing. It does not need to store the keys of children nodes in each traversed node. While it still has to record the paths of children, this information can be compressed because they are only required to be *nearly random instead of specified by application*, as introduced in Section 3.1. Therefore, if we use the same block size, the node of a data-independent tree can include more children than that of the data-dependent tree in prior OMAPs.

Conceptually, our framework needs only an ORAM here for storing data hash information. It is not necessary to use the recursive ORAM and establish the data-independent tree. Applying some more advanced (but impractical) hierarchy ORAMs [5, 63] with optimal complexity, our framework can achieve better complexity than the constructions in this paper, e.g., with the ORAM in [5] and group OMAPs in Section 6, the communication bandwidth can be as low as $O(\log n \log\log n)$ instead of $O(\log^2 n/\log\log n)$ in our constructions. However, we focus on the practicality of OMAPs in realistic scenarios instead of only the theoretical complexity, and up to now, only the recursive tree-based ORAMs [32, 47, 72] have been demonstrated the practicality under $O(\log n)$ client-side storage. To this end, we adopt the recursive ORAM and compress the data-independent tree to improve the ORAM performance.

Group OMAPs. The data-independent tree seems nice but is theoretically equivalent to a simple hash table allowing collisions. There can be a group of KV pairs mapped to the same leaf node in the tree. To this end, after we find the leaf node in the ORAM, we still need to access the required pair within this group via existing OMAPs. Take the OMAP based on ODS+AVL [79] as an example, we establish a new ODS and then organize each non-empty group as an AVL tree stored in this ODS. The height of each AVL tree is $O(\log\lambda)$ as each group has at most $O(\lambda)$ pairs [27, 65]. To access a pair in a group, we traverse only $O(\log\lambda)$ nodes of the AVL tree, and corresponding interaction rounds are also $O(\log\lambda)$.

We still need to clarify how we combine the data-independent tree and group OMAPs. There are two ODSs separately for the two components. Recall we need to know the path of the root node of an AVL tree for traversing the tree. So before we look up the pair required within the ODS for the corresponding group, we first find the path of the root from the ODS for the data-independent tree, i.e., the variable pt_j for group $\mathcal{G}_j$. Until now, we smoothly integrate the two components together for constructing new more efficient OMAPs. We sum the two components for calculation. We conclude the interaction rounds as $O(\log n/\log\log n)+O(\log\lambda)$ and the communication bandwidth as $O(\log^2 n/\log\log n)+O(\log n\log\lambda)$. Note that the ODS for the data-independent tree can be built by the existing recursive ORAM where $X \sim O(\log n/\log\log n)$ [47] and the OMAP for groups can be constructed by any existing OMAP constructions [15, 20, 30, 67, 79]. *So up to now, we can design five OMAP constructions which are asymptotically better than prior works.*

5 DATA-INDEPENDENT TREE

In this section, we propose a new ORAM protocol named **De-a**mortized ORAM (DAORAM). It is motivated by addressing the impracticality of the prior recursive ORAMs [32, 47]. As we pointed out in Section 3.1.3, while the recursive ORAM in [47] can be used to instantiate new OMAPs with better complexity under our framework, it is impractical in production. This makes the OMAPs based on it noncompetitive to prior OMAPs for real-world applications. To this end, we propose DAORAM to address all the shortages of [47] presented in Section 3.1.3 and make our OMAPs indeed practical. For brevity, we still treat the recursion process as traversing a complete X-ary tree and call each KV pair as a node in the tree.

5.1 Construction

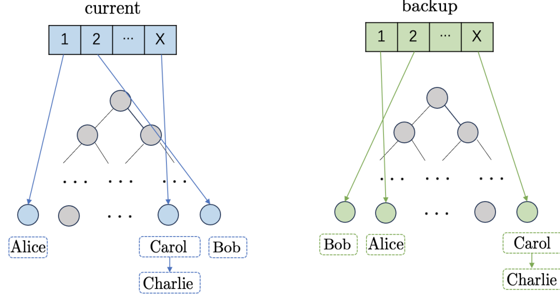
In this section, we propose a new recursive ORAM protocol named **De-a**mortized ORAM (DAORAM) for achieving the ODS of the data-independent tree efficiently. Motivated by our observations in Section 3.1.3, there are three design goals for our new protocol:

(1) **De-amortization:** It should perform stably, even the worst-case performance is still efficient.
(2) **Larger X:** It should enables a large X to reduce interaction rounds as much as possible.
(3) **Perfect reset:** There is no repeated GC||IC with *probability 1* whatever the database size is.

Overall, compared with the fixed approach [47], our new protocol is expected to be more practical and efficient.

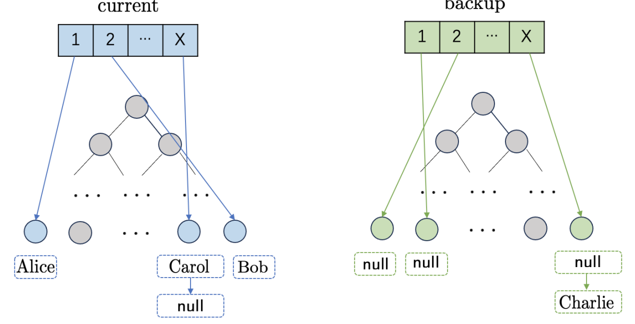
Reset analysis. Here we explain why the reset operation is expensive and should be "removed". First of all, lots of works [9, 34, 60, 66, 80] have demonstrated that the main cost overhead of ORAMs is *communication* including the interaction rounds and communication bandwidth. That's why a line of work [6, 21, 34, 63, 66] are trying to pursue better interaction rounds and communication bandwidth. The reset operation becomes costly as it needs C to download X paths and then place them back. Although C can retrieve μ paths in parallel (within the same interaction round) to reduce interaction rounds, it needs to provide $O(\mu\log n)$ storage and still interacts with $\mathcal{S}$ for X/μ rounds. Recall we always assume C owns only $O(\log n)$ storage to cover a wide range of devices like smartwatches, which implies μ should be a constant. So the reset cannot avoid $O(X)$ interaction rounds and transferring X paths between C and $\mathcal{S}$. Worse more, C is often assumed to interact with $\mathcal{S}$ **under WAN** [12, 20, 45, 67] where the latency can be high. This makes the multiple interaction rounds in the reset further unacceptably expensive, becoming one bottleneck for real-world applications where low latency is important [34].

Intuition. Now we can introduce the intuition of our construction. The main challenge is how to remove the reset while preserving the efficiency of each query, i.e., each query processing should be as nearly fast as the baseline in the fix, i.e., no reset happens during the query processing. As we have analyzed that the expensive



A query Q requests item change "$Carol \rightarrow Charlie$" in both the two groups.

Figure 4: The prior strategy with a query "*Carol* ⟶ *Charlie*".



A query Q : "$Carol \rightarrow Charlie$" preserves the data in only one group.

Figure 5: Our new strategy with a query "*Carol* ⟶ *Charlie*".

costs in ORAM come from interaction rounds and communication bandwidth, we address them with the following guideline:

(1) Removing the $O(X)$ interaction rounds for the reset;
(2) Transferring the X paths partially in each query, e.g., transferring only one of the X paths in each query.

In this way, C will feel only a little sacrifice on efficiency because the overhead brought by (2) is very cheap relative to the total time usage. But the practicality is improved much as lots of interaction rounds incurred by resets are not needed any more. To achieve the guideline above, we resort to the de-amortization algorithm to *"spread" the reset operation over many queries*. Specifically, in this paper, we make use of the interaction rounds in usual query processing to partially transfer reset paths and process the reset.

De-amortization philosophy. De-amortization is a classic topic about ORAM and has been studied a lot [7, 17, 54, 61]. However, all prior works aim to hierarchy ORAMs [5, 64] and cannot be non-trivally applied in tree-based ORAMs. This is because among tree-based ORAMs, only the works above [32, 47] which try to compress children within a node suffer from the worst-case performance. However, such ORAM protocols are important for our framework. As far as we know, this work is the first to introduce de-amortization in tree-based ORAMs and adopt a philosophy different from prior works, which can be helpful for understanding the recursive ORAM and OMAPs based on it.

When we try to follow the philosophy of prior de-zmortizations [7, 17, 54, 61], we found that can be much expensive in tree-based ORAMs, which motivates us to present a new and more useful strategy for that in tree-based ORAMs. The prior works hope to prepare a backup for the expensive operation (like the reset) during usual queries. So if C needs reset, it directly starts with the backup and then prepares the next backup. However, this always brings copies of data such that in the end we have to execute de-duplication to delete data copies, which is the main bottleneck in prior works. Moreover, to guarantee data consistency, C has to execute each update query in both the currently used data and backup data which we call as the *current group* and *backup group*, respectively. As shown in Figure 4, C separately completes the update query: "*Carol* ⟶ *Charlie*" within the two groups, multiplying the query costs by 2. In this paper, we propose a new lazy strategy for tree-based ORAMs: *pursue the expensive operation instead of preparing it in advance.* In short, if C needs to reset the block when accessing a pair, it just directly **resets this pair alone** and resets all other pairs in this node during the next usual queries. This avoids data copies and also de-duplication, enabling more efficient ORAM construction. The data consistency under our strategy is also guaranteed in a more efficient way. We always preserve each item in only one group. When the reset is triggered by the update query: "*Carol* ⟶ *Charlie*, we remove this item from the current group and put the updated item in the backup group, which requires only **one** usual query cost in tree-based ORAM. We remark that this is because although we depict two trees in figures for ease of understanding, the two trees can be integrated into one in storage, allowing the removal and placement to be completed in the same query process. The challenge is to let C always know which group to find for each data, e.g., if the next query is to search *Charlie*, then C should find it in the backup group instead of the current group for correctness. Besides, C needs to remove all items from the current data group to the backup group before the next reset is triggered for continuous de-amortization. Our construction proposed below will address the two challenges with practicality.

Data structures. We first define the data structure in an internal node of the data-independent tree. Suppose this node containing the paths of children with key $\{a, a+1, ..., a+X-1\}$, we store two groups of counters: they are the compression for reset and pursuing reset denoted by G^r and G^p:

$$\mathsf{G}^r : \mathsf{GC}^r||\mathsf{IC}_0^r||\mathsf{IC}_1^r||\mathsf{IC}_2^r||\cdots||\mathsf{IC}_{X-1}^r,$$

$$\mathsf{G}^p : \mathsf{GC}^p||\mathsf{IC}_0^p||\mathsf{IC}_1^p||\mathsf{IC}_2^p||\cdots||\mathsf{IC}_{X-1}^p.$$

Consistent to prior works [32, 47], we let GC occupies α bits and IC occupies γ bits such that $\alpha + X \cdot \gamma \sim O(\log n)$. Besides, we **additionally add one bit** $\mathsf{b} \in \{0, 1\}$ as the indicator variable. The path calculation for the child with key $a + j$ ($j \in [X]$) based on the two groups are as below:

$$pt_j^r = \mathsf{PRF}_{\mathsf{K}^{\mathsf{b}}}(a + j||\mathsf{GC}^r||\mathsf{IC}_j^r), \quad (2)$$

$$pt_j^p = \mathsf{PRF}_{\mathsf{K}^{1-\mathsf{b}}}(a + j||\mathsf{GC}^p||\mathsf{IC}_j^p) \quad (3)$$

where $(\mathsf{K}^0, \mathsf{K}^1)$ are two secret keys for PRF. That means we define two different calculations for the two groups and we will show how they are useful in query processing.

Query Processing. Now we describe the specific query processing with three phases as below.

(1) *The initialization phase* happens only once in the beginning. Similar to prior works [32, 47], it initializes $(\mathsf{G}^r, \mathsf{G}^p)$ in an internal node of the data-independent tree to include paths for the children with keys $\{a, a+1, ..., a+X-1\}$. The initial values

are set as below:

$$\forall j \in [X], \mathsf{IC}^r_j := 1,\ \mathsf{IC}^p_j := 0.$$

Also $(\mathsf{GC}^r, \mathsf{GC}^p)$ and b are set as 0. The child with keys $a+j$ is guaranteed to be placed in the path calculated by Equation 2.

(2) *The query phase* is for processing a query from C. Here we describe how the access is done between an internal node and its children. C repeats this process for $O(\log n/\log X)$ internal nodes to traverse the data-independent tree. Suppose C wants to access the node with key $a+j$ ($j \in [X]$) in level i, and it has got the internal node in level $i-1$ whose children own keys $\{a, a+1, ..., a+X-1\}$. Then C calculates (pt^r, pt^p) according to Equation 2 and Equation 3. Now C execute procedures according to the value of IC^r_j:
   (a) If $0 < \mathsf{IC}^r_j < 2^\gamma - 1$, C directly retrieves the child node using pt^r. Then C increments $\mathsf{IC}^r_j := \mathsf{IC}^r_j + 1$ to calculate the new assigned path to this child node with Equation 2.
   (b) If $\mathsf{IC}^r_j = 2^\gamma - 1$, i.e., it cannot be incremented more for obliviousness. Next C sets $\mathsf{GC}^p = \mathsf{GC}^p + 1$, $\mathsf{IC}^p_j = 1$. Then C retrieves the child node using pt^r and assigns the node with the new path pt^p to place it back. Finally, C sets $\mathsf{b} = 1 - \mathsf{b}$, $\mathsf{IC}^r_j = 0$, and then swaps $(\mathsf{G}^r, \mathsf{G}^p)$, i.e., the original reset group now needs to be the one for pursing reset because one value in it reached the upper bound.
   (c) If $\mathsf{IC}^r_j = 0$, then C uses pt^p_j to retrieve the node. Next it increments $\mathsf{IC}^r_j := \mathsf{IC}^r_j + 1$ to calculate the new assigned path to the node with Equation 2. After that, it sets $\mathsf{IC}^p_j = 0$.

   With steps above, C identify the retrieved path and new assigned path so it can write back the child node.

(3) *The reset phase* is executed in parallel with the query phase to reuse the interactions in the query phase. There are two cases:
   (a) If all IC^rs except IC^r_j are non-zeo, then C just retrieves a random path, evicts it and wirtes it back.
   (b) If there exists $j_1 \neq j$ such that $\mathsf{IC}^r_{j_1} = 0$, then C accesses the child node with key $a + j_1$ identically to case (c) in the query phase.

During the execution, every time C issues a query, it interacts with $\mathcal{S}$ to execute the query phase and reset phase in parallel. In total, C will retrieve 2 paths, process them, and return them. The two path are retrieved within the same one interaction round. This guarantees that $\mathsf{GC}||\mathsf{IC}$ strictly increases if $2^\gamma > X$, the correctness proof is provided in Appendix A.1.

Remark. The readers may notice that it is not easy for C to always distinguish G^r and G^p correctly because they have the same format and value range. So we will always place G^p behind G^r. The indicator bit b is exactly used to mark which secret key corresponds to the first group.

Reducing groups. Now we have achieved the de-amortization with two compressed groups within an internal node but there is only one group in prior works [32, 47]. Next, we show how to optimize our construction for reducing group numbers. The setting of two-group parameters helps understand how the reset is partially done per access. But the two groups can be integrated based on a non-trivial observation: for any $j \in [X]$, it holds that one of $(\mathsf{IC}^r_j, \mathsf{IC}^p_j)$ must be zero and the other is non-zero. So we can record only the non-zero value and use only a bit to imply which groups it belongs to. Besides, as GC^r and GC^p can be repeated without sacrificing security, we replaced them with only one variable GC. Now we do the increment $\mathsf{GC} := \mathsf{GC} + 1$ every two swaps, i.e., both the logic GC^r and GC^p have been used for recursion. We use the variable b to do this: each time swap happens and also $\mathsf{b} = 1$, we increment GC. Finally, the data structure within a node is:

$$\mathsf{GC}||\mathsf{IC}_0||\mathsf{IC}_1||\cdots||\mathsf{IC}_{X-1} \text{ and } \mathsf{g}_0||\mathsf{g}_1||\cdots||\mathsf{g}_{X-1}||\mathsf{b}$$

where $\mathsf{g}_j \in \{0,1\}$ implies if IC_j belongs to G^r and $\mathsf{b} \in \{0,1\}$.

5.2 Analysis

In this section, we mainly give the analysis of performance to show DAORAM indeed achieves all three design goals in Section 5.1. The proof of correctness and security is formally given in Appendix A.1. For the performance, we analyze three metrics including *interaction round*, *interaction bandwidth*, and *computational complexity* per query. In DAORAM, the interaction round is $O(\log n/\log\log n)$ as we still enable $X \sim O(\log n/\log\log n)$ and assume $O(\log n)$ client-side storage. The communication bandwidth is calculated as:

$$O(\log X + \log X^2 + \cdots + \log n) = O(\log^2 n/\log X).$$

where $\log X^i$ denotes the communication bandwidth in the ith interaction round. So when X is $O(\log n/\log\log n)$, the communication bandwidth is $O(\log^2 n/\log\log n)$. To complete the calculation about retrieving a node in ith level, the main computation is sorting the union of $O(\log X^i)$ retrieved nodes and $O(\log X^i)$ nodes in the stash, which requires $O(\log X^i \log\log X^i)$ computation. So the total complexity is

$$O(\log X \log\log X + \log X^2 \log\log X^2 + \cdots + \log n \log\log n)$$

which can be bounded by $O(\log^2 n \log\log n/\log X)$. When X is $O(\log n/\log\log n)$, the total complexity is $O(\log^2 n)$.

It is easy to notice that we successfully remove all interaction rounds for resets by applying the interaction rounds in usual query process. The cost is that we transfer two paths in each access while prior works transfer only one, i.e., we increase one path communication, which is very cheap for the whole query processing. So we achieve the **de-amortization** design goal as expected. Now we explain our de-amortization naturally supports the second and third design goals. For parameters, the correctness and security of DAORAM require only $2^\gamma > X$ even when n whatever n is. This is much more relaxed than that in [47] (cf. Observation 2). Therefore, we can further enlarge X as much as possible to minimize the interaction round for actual performance by solving the following equations to get values of (γ, X):

$$\begin{cases} 2^\gamma - 1 = X \\ \alpha + X \cdot \gamma \sim O(\log n) \end{cases} \tag{4}$$

6 GROUP OMAP

In this section, we introduce the group OMAP under our framework. Its design is creatively modified from existing OMAPs to fit our framework. Under our framework, n KV pairs are randomly divided into n groups. Prior works [35, 72] conclude that each group has at most $O(\lambda)$ items. However, the recent work [27] proposes a theorem that shows this bound can be very low in practice, which makes our method truly more practical than prior OMAPs. For example, given $\lambda = 128$, $n = 2^{24}$, there exists a group consisting of more than

52 pairs with a probability no larger than 2^{-128}. Here we introduce the simplified theorem in [27] here for completeness:

Theorem 6.1. *With n items independently and uniformly randomly mapped to one of n groups, then for the following function $f(n,\lambda)$ that outputs the bound, the probability of there exists one group consisting of more than $f(n,\lambda)$ is negligible in λ.*

$$f(n,\lambda) = min(n, exp[W_0(e^{-1}(\log n + \lambda - 1)) + 1])$$

where $W_0(\cdot)$ is branch 0 of the Lambert W function.

We list a table to identify the small value of $f(n,\lambda)$ under $\lambda = 128$ in Appendix B. Although we continue using $O(\lambda)$ to bound the group size, the readers should realize this bound can be small enough to be efficient. Now we describe the design and performance of the group OMAP. Recall it is used to store the n groups and access any pair within a group obliviously. For obliviousness, it is required:

- *Group obliviousness:* $\mathcal{S}$ cannot infer which group among the n groups is accessed during the query processing.
- *Pair obliviousness:* $\mathcal{S}$ cannot infer which pair among the $O(\lambda)$ pairs within the group is accessed in the query processing.

OMAPs as a whole. To achieve the pair obliviousness, we can apply any existing OMAP [15, 30, 67, 79] to process the $O(\lambda)$ pairs in the same group. The essence is to guarantee the group obliviousness: *we store all the n groups within the same ODS tree for access.* Take the OMAP based on ODS for the AVL tree as an example, pairs in the same group are organized as an AVL tree. Then the n AVL trees are stored in the same ODS tree. To access a pair in the AVL tree, we just traverse the AVL tree obliviously via the ODS tree. Recall to access the AVL-based OMAP, we need to know how to find the root node of the AVL tree, which is provided by the DAORAM. In the end, the root is updated and rewritten to the DAORAM. As most existing OMAPs [15, 30, 67, 79] are based on a search tree, we can in general apply such a process to all of them for different trade-offs.

Here we introduce three OMAP approaches [15, 30, 79] we use with DAORAM under our framework to instantiate three new specific OMAP constructions. They adopt ODS for different search trees with various trade-offs. We summarize them as below and refer their complexity description to Table 1.

- **ODS+AVL:** This OMAP [79] is based on *oblivious AVL tree*. It is the simplest and achieves the best bandwidth blowup. However, it is the most expensive in reality as pointed out by [67]. Notably, it guarantees the insertion and search are indistinguishable.
- **ODS+AVL*:** This OMAP [15] is also based on *oblivious AVL tree* and is the SOTA work in VLDB 2024. It allows search to be distinguishable from insertion and further optimizes search for better efficiency. It also contributes to the client-side oblivious algorithm as it is implemented in Intel SGX. As our works focus on the client-server setting [75] where the client-side algorithms are not required for obliviousness, we just adopt more efficient algorithms for this OMAP in $\mathcal{C}$.
- **ODS+B+:** This OMAP [30] is based on *oblivious B+ tree* and thus allows a lower tree height and fewer interaction rounds. However, it achieves the reduced interaction rounds at the cost of communication bandwidth: compared to prior methods [15, 79], the block and bucket size here have to be extended for storing more keys within one node of the B+ tree.

Reset number		0	1	2	3
Proset	time (s)	0.29	1.72∼2.95	3.14∼4.75	4.55∼5.06
	band (KB)	24	91.5∼392.3	234.2∼685.4	452.1∼903.3
Fixset	time (s)	**0.37**			
	band (KB)	**39.7**			

Table 3: Comparison between Fixset and Probset on stability.

7 EVALUATION

Our proposed new framework for designing OMAP comprises two important components: an ORAM for *the data-independent tree* and an ODS for *group OMAPs*. We present a novel ORAM protocol named DAORAM which not only surpasses the performance of prior solutions [32, 47] suited for the data-independent tree, but also achieves *de-amortization* for practicality. For the ODS, we adapt existing OMAP schemes described in Section 6. To this end, we combine DAORAM with the three aforementioned OMAPs to build three new OMAP constructions. In this section, we conduct experiments to study the following two questions:

Q1. What is the performance gain of DAORAM compared to prior ORAMs [32, 47] for the data-independent trees? (Section 7.1)

Q2. What is the performance gain of our new OMAPs compared to prior OMAPs [15, 30, 67, 79]? (Section 7.2)

Settings. Consistent with a line of prior works [15, 30, 67, 79] on ORAM/OMAP, we implement our ORAM/OMAPs in a client-server setting. $\mathcal{S}$ operates a powerful machine, featuring an Intel Xeon Platinum 8160 CPU (96 cores, 2.10 GHz) and 376 GB memory, located in Hangzhou, China. $\mathcal{C}$ operates on a relatively lightweight Alibaba Cloud machine equipped with 4 vCPUs (from an Intel Xeon Platinum 8269CY, 2.50GHz) and 16 GB memory and located in Beijing, China. Importantly, $\mathcal{C}$ and $\mathcal{S}$ interact with each other over the WAN to simulate reality, with a bandwidth of 100 Mbps and an average latency of 38 ms. Our constructions are implemented in Python 3.10, where the encryption (AES 128-bit) and PRF are imported from the pycryptodome package [3]. To ensure a fair comparison, we implement prior OMAPs following their open-sourced repositories [1, 2] in Python. We follow the commonly used parameters to set up ORAMs: each bucket has 4 blocks, and the block size of DAORAM is set to 512 bits, consistent with [32]. Following prior works [27, 30, 59], we primarily conduct experiments on synthetic datasets, as the obliviousness property guarantees that ORAMs/OMAPs perform independent of data distributions [12, 42]. We generate the synthetic datasets of varying sizes to evaluate the scalability of our proposed constructions.

7.1 Practical ORAM with de-amortization

To demonstrate that DAORAM is the most practical and efficient, we implement DAORAM and two other recursive ORAMs for comparison. We refer to the ORAM constructions by the names listed below in the following discussions:

- Freeset: Freecursive [32] serves as the baseline, although it does not achieve obliviousness. It represents the best possible average processing time for queries, as it requires the fewest resets.
- Probset: Freecursive with the probabilistic reset [47] is the only ORAM (before ours) that satisfies both the obliviousness and the data-independent tree requirements. However, it is impractical due to its inefficiency and unstable query performance.

- Fixset: Our DAORAM with the fixed reset not only ensures stable performance but also performs more efficiently than Probset.

De-amortization. We compare Fixset and Probset on their query performance. We run both of them on a synthetic dataset with 2^{24} (over 16,000,000) KV pairs, where both key and value are 4 bytes, and perform 2^{20} queries. Since Probset processes queries with random resets, we categorize its queries based on the number of resets that occur during the query. The corresponding bandwidth and processing time per query are shown in Table 3. The results are intervals when the reset number is non-zero because resets can occur in ORAMs of different sizes, leading to varying costs. Clearly, the costs noticeably increase even if the reset number is only 1. The processing time can be $6 \sim 17\times$ slower than that in the best case, where the reset number is 0. In contrast, Fixset always performs stably, and the results show that its performance is comparable to even the best-case performance of Probset. Additionally, we evaluate the stash size stored in C to demonstrate DAORAM also outperforms prior works [32, 47] in stash size. We show the maximum stash size used under different query numbers in Figure 6. While the query distribution does not affect the reset operations, it possibly impacts the stash sizes. To study this, we generate queries under two typical query distributions: 1) all queries *repeatedly* access the same pair, and 2) queries follow *uniform* distributions. It is shown that Fixset has an impressively smaller clientside stash than the other two protocols. The stash size in Fixset is only one-third of that in Probset! This results from two advantages of DAORAM brought by de-amortization. Firstly, there are dummy accesses in DAORAM, which only evicts a random path. They help DAORAM reduce the stash size but the other two works do not have such dummy accesses. Secondly, while the other two protocols retrieve only one path per access, DAORAM retrieves two paths per access corresponding the query and reset phase, respectively. This allows more aggressive eviction strategies, reducing the stash size. All these results demonstrate, benefited from de-amortization, Fixset is the most practical among the protocols of interest.

Efficiency. We also compare the average processing time and communication time of all queries to evaluate their performance. With each of the three ORAM protocols, we run 2^{10} queries on databases with sizes ranging from 2^{10} to 2^{24} and present the results in Figure 7. The queries are generated without repetition to ensure Fixset maintains its best-case performance by avoiding any resets. Despite this, Fixset's performance remains close to that of Freeset and improves upon Probset by $21\% \sim 47\%$. Thus, Freeset (i.e.,DAORAM) is highly suitable for production due to its efficiency improvements and practicality.

7.2 Efficient OMAP with less communication

In Section 6, we list three existing OMAP constructions. Under our framework, we construct three new OMAPs using DAORAM with each of these OMAPs. We compare the efficiency of each new OMAP with the corresponding existing OMAP it is based on to demonstrate our framework accelerates OMAPs:

- DAORAM+AVL vs. ODS+AVL: ODS+AVL [79] is the baseline and is included in the comparison as it is the first and most widely-used OMAP [12, 46, 59].

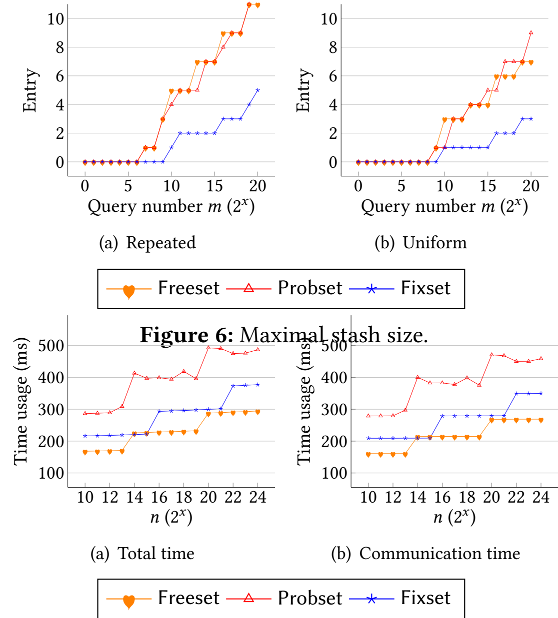


Figure 6: Maximal stash size.

Figure 7: Amortized cost per query.

- DAORAM+AVL* vs. ODS+AVL*: ODS+AVL* [15] is the state-of-the-art OMAP based on the AVL tree. It optimizes the search algorithm of the baseline for better efficiency.
- DAORAM+B+ vs. ODS+B+: ODS+B+ [30], benefiting from its large branching degree, is the most efficient OMAP to date.

As claimed in [75] and Section 5.1 of this work, the performance bottleneck of OMAPs in a client/server setting is the bandwidth and interaction rounds. Therefore, we use the following three metrics to evaluate OMAPs: *1) the time usage of operations, 2) the number of interaction rounds, and 3) the communication bandwidth.*

Insertion. We run all six OMAPs mentioned above on databases with sizes ranging from 2^{10} to 2^{24} and execute 100 queries, as in [30, 67]. Here we mainly focus on the insertion operation. Insertion is essential to OMAPs because it captures the write operations on databases and in the original and ideal OMAPs [79], even the search operation should seem identical to insertion. So we present the insertion comparison here and leave the search comparison in Appendix D.1 for space. The time usage of insertion is depicted in Table 4 and we decompose the time in detail to show the speedup in different components further. Besides, to explain the speedup, we list the improvement in communication of our OMAPs in Table 5.

As shown in the tables, there is a substantial speedup of our OMAPs to prior OMAPs. Firstly, in communication time (T_2 in Table 4), our OMAPs achieve a speedup of $37.0\% \sim 72.0\%$ compared to the corresponding OMAPs they are based on. This results from the reduced interaction rounds and bandwidth. We significantly reduce the interaction rounds and bandwidth with a speedup from $35.6\% \sim 92.6\%$! The interaction round is the dominant factor in communication as the OMAPs are run under WAN, hence the speedup of T_2 is closer to that of the interaction round. Also, communication occupies the most time usage during query processing, our OMAP mainly improves communication to enhance efficiency. Secondly, our OMAPs also speed up the client-side calculation (T_1 in Table 4) between $33.3\% \sim 71.1\%$ although T_1 owns only a very minor proportion. This speedup comes from reduced bandwidth, which

n		2^{10}			2^{13}			2^{16}			2^{19}			2^{21}			2^{24}		
Time components		T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3
AVL	prior (s)	0.06	2.91	2.97	0.09	3.72	3.81	0.17	4.70	4.87	0.27	5.51	5.78	0.31	6.12	6.43	0.38	6.94	7.32
	ours (s)	0.04	1.25	1.31	0.06	1.48	1.55	0.07	1.58	1.64	0.09	1.81	1.89	0.10	1.82	1.93	0.11	1.94	2.05
	speedup (%)	33.3	57.0	55.9	33.3	60.2	59.3	58.8	66.4	66.3	66.7	67.2	67.3	67.7	70.3	70.0	71.1	72.0	72.0
B+	prior (s)	0.04	1.46	1.57	0.05	1.79	1.84	0.08	2.01	2.09	0.10	2.51	2.61	0.13	2.73	2.86	0.16	2.95	3.11
	ours (s)	0.02	0.92	0.94	0.03	0.94	0.97	0.04	0.98	1.02	0.04	1.00	1.04	0.05	1.04	1.09	0.06	1.08	1.14
	speedup (%)	50.0	37.0	40.1	40.0	47.5	47.3	50.0	51.2	51.2	60.0	60.2	60.2	61.5	61.9	61.9	62.5	63.4	63.3

Table 4: Time usage of insertion. T_1 is calculation time, T_2 is the communication time, and T_3 is the total processing time.

n		2^{10}		2^{13}		2^{16}		2^{19}		2^{21}		2^{24}	
Communicate		round	band (KB)	round	band (KB)	round	band (KB)	round	band (KB)	round	band (KB)	round	band (KB)
AVL	prior (s)	90	345.6	114	554.49	144	884.74	168	1204.22	186	1476.10	210	1881.60
	ours (s)	58	128.51	58	130.05	60	132.86	60	135.17	60	136.70	62	139.26
	speedup (%)	35.6	62.8	49.1	76.5	58.3	85.0	64.3	88.8	67.7	90.7	70.5	92.6
B+	prior (s)	42	75.26	54	124.42	66	185.86	72	221.18	84	301.06	96	393.22
	ours (s)	28	28.67	28	30.21	30	33.02	30	35.33	30	36.86	32	39.42
	speedup (%)	33.3	61.9	48.1	75.7	54.5	82.2	58.3	84.0	64.3	87.8	66.7	90.0

Table 5: Interaction rounds and communication bandwidth of insertion.

implies we retrieve fewer items to calculate, thus T_1 decreases. The speedup of T_1 is lower than the reduction in bandwidth because the client-side calculation in our OMAPs is more complex, lowering the speedup. Finally, in the whole query processing time (T_3 in Table 4), our OMAPs achieve a speedup of 40.1% ∼ 72.0%. The most efficient of our OMAPs is DAORAM+B+, which is almost 6× faster than the baseline ODS+AVL for insertions. We remark our speedup comes from the lower asymptotic complexity, both the two tables show with n increasing, the speedup is more and more significant. So it is predictable the speedup of our OMAPs on insertions will be more pronounced as the database size increases due to the superior complexity of our OMAPs, i.e., our OMAPs are expected to perform even better in very large databases in production.

Extended experiments. There are more extensive experiments conducted to evaluate our ORAM and OMAPs, which are shown in the appendix for space. For DAORAM, we test the impact of different accesses on its stash size, validate its obliviousness by running it under multiple query distributions, and compare it with prior tree-based ORAMs to provide further insights. The results are presented in Appendix A.3. For OMAPs, we evaluate more of its operations (e.g., search, delete, etc), validate its obliviousness with six distinct query distributions with different skewness and two different datasets, and finally evaluate it under network conditions across a wide range of latencies. These results are provided in Appendix D.

8 RELATED WORK

The leakage from access patterns has been widely recognized as dangerous in EDBs, prompting handful of works presented in the communities of databases [20, 30, 52], security [59, 67, 79], and cryptography [5, 7, 64] to achieve *obliviousness* in EDBs. Our work is closely aligned with two well-known areas: *Oblivious RAM* (ORAM) and *oblivious data structure* (ODS).

ORAM. ORAM is an essential primitive that counters attacks based on access pattern leakage, with extensive research in various directions [5, 7, 47, 72, 73]. Our works follow a line of works [13, 32, 47, 66, 78] to improve the relatively practical *tree-based* ORAM. Tree-based ORAMs can be divided into recursive ORAMs [32, 47, 73] and non-recursive ORAMs [78, 81]. Non-recursive ORAMs use $O(n)$ client-side storage to enhance their efficiency, but this storage requirement may be infeasible in production [32, 75]. Recursive ORAMs require small client-side storage, typically $O(\log n)$, making them more practical. However, as discussed in Section 2, one data access in recursive ORAMs involves $O(\log n)$ interaction rounds, which can be expensive, especially over WAN. Our work follow [32, 47] to significantly reduce the number of interaction rounds to $O(\log n/\log\log n)$. Notably, our proposed scheme DAORAM elegantly avoid the costly worst-case performance in prior works [32, 47]. To our knowledge,DAORAM is the most efficient and practical recursive ORAM protocol to date.

ODS. Since Wang et al. [79] introduced the concept of ODS, extensive research has focused on exploring and improving ODS constructions. Wang et al. [79] propose techniques and constructions for a variety of classic data structures, including trees, sets, and graphs. In particular, they provide the first construction for the oblivious map (OMAP), using an oblivious AVL tree, which is broadly adopted by many EDBs [12, 28, 36] and serves as the baseline for comparisons in this work. Following Wang et al., Roche et al. [67] propose a new tree structure named HIRB (similar to a B tree) to establish a more efficient OMAP. Currently, the SOTA works are [15, 30], which adopt an oblivious B+ tree and an optimized AVL tree, achieving the best performance to date. However, the design philosophy of search trees causes these constructions to be limited by the $O(\log^{1.5} n)$ communication bandwidth lower bound of oblivious search trees, as proven by [48]. Moreover, constructions of search tree based OMAPs have not yet overcome

the $O(\log^2 n)$ bandwidth. As far as we know, we are the first to adopt a framework other than the oblivious search tree and achieve $O(\log^2 n/\log\log n)+O(\log\lambda\log n)$ communication bandwidth with OMAP constructions.

Our work on ORAM in this paper also suggests that oblivious hash tables [79] can have a similar bandwidth complexity, but as we discussed in Section 1, this approach is still more costly than our constructions. Enigma [75] is another study on OMAP that is independent of our research, as it focuses on optimizing the performance when implementing OMAP in secure enclaves, e.g., the page swaps between inside and outside the enclave. In addition, several other works address OMAP in secure enclaves, with a strong emphasis on achieving obliviousness within the enclave (i.e., the obliviousness in C). All of these works can benefit from our OMAP, as it improves the performance of oblivious algorithms in $\mathcal{S}$. We leave it as future works to integrate our work with prior algorithms in C to achieve obliviousness practically in both C and $\mathcal{S}$, a concept referred as *double-obliviousness* in Oblix [59].

9 CONCLUSION

In this paper, we propose a new framework for designing a fundamental oblivious data structure in encrypted databases: *oblivious map* (OMAP). We are the first to combine the *oblivious hash table* (which allows collisions) with an oblivious search tree to build more efficient OMAPs. We propose a new ORAM protocol named DAO-RAM, which is the most efficient and practical recursive tree-based ORAM, for the oblivious hash table. By combining the oblivious hash table with three prior OMAPs based on search trees [15, 30, 79], we present three OMAP constructions and empirically demonstrate that they significantly outperform prior OMAPs. Our work can enhance the efficiency of all encrypted key-value databases and more general encrypted databases.

REFERENCES

[1] [n.d.]. ObliDB open-sourced repository. ([n. d.]). https://github.com/SabaEskandarian/ObliDB

[2] [n.d.]. An open-sourced repository for ODS+AVL. ([n. d.]). https://github.com/obliviousram/oblivious-avl-tree

[3] [n.d.]. Python package pycryptodome. ([n. d.]). https://github.com/Legrandin/pycryptodome

[4] Yuriy Arbitman, Moni Naor, and Gil Segev. 2009. De-amortized Cuckoo Hashing: Provable Worst-Case Performance and Experimental Results. In *Automata, Languages and Programming*, Susanne Albers, Alberto Marchetti-Spaccamela, Yossi Matias, Sotiris Nikoletseas, and Wolfgang Thomas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 107–118.

[5] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. 2020. OptORAMa: optimal oblivious RAM. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II 30.* Springer, 403–432.

[6] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. 2022. Optimal Oblivious Parallel RAM. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* SIAM, 2459–2521.

[7] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, and Elaine Shi. 2023. Oblivious RAM with worst-case logarithmic overhead. *Journal of Cryptology* 36, 2 (2023), 7.

[8] Alexander Bienstock, Sarvar Patel, Joon Young Seo, and Kevin Yeo. 2023. Near-Optimal Oblivious Key-Value Stores for Efficient PSI, PSU and Volume-Hiding Multi-Maps. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 301–318. https://www.usenix.org/conference/usenixsecurity23/presentation/bienstock

[9] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. 2015. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* 837–849.

[10] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. 2013. Data-oblivious graph algorithms for secure computation and outsourcing. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security.* 207–218.

[11] Dmytro Bogatov, Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2021. εpsolute: Efficiently Querying Databases While Providing Differential Privacy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security.* 2262–2276.

[12] Xinle Cao, Yuhan Li, Dmytro Bogatov, Jian Liu, and Kui Ren. 2023. Secure and Practical Functional Dependency Discovery in Outsourced Databases. *Cryptology ePrint Archive* (2023).

[13] Anrin Chakraborti, Adam J Aviv, Seung Geol Choi, Travis Mayberry, Daniel S Roche, and Radu Sion. 2019. rORAM: Efficient Range ORAM with O (log2 N) Locality.. In *NDSS.*

[14] Anrin Chakraborti and Radu Sion. 2016. POSTER: ConcurORAM: High-Throughput Parallel Multi-Client ORAM. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16).* Association for Computing Machinery, New York, NY, USA, 1754–1756. https://doi.org/10.1145/2976749.2989062

[15] Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2023. GraphOS: Towards Oblivious Graph Processing. *Proc. VLDB Endow.* 16 (2023), 4324–4338. https://api.semanticscholar.org/CorpusID:265455667

[16] TH Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. 2018. Cache-oblivious and data-oblivious sorting and applications. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms.* SIAM, 2201–2220.

[17] T-H Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. 2017. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23.* Springer, 660–690.

[18] Zhao Chang, Dong Xie, and Feifei Li. 2016. Oblivious RAM: A dissection and experimental evaluation. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1113–1124.

[19] Zhao Chang, Dong Xie, Feifei Li, Jeff M. Phillips, and Rajeev Balasubramonian. 2022. Efficient Oblivious Query Processing for Range and kNN Queries. *IEEE Transactions on Knowledge and Data Engineering* 34, 12 (2022), 5741–5754. https://doi.org/10.1109/TKDE.2021.3060757

[20] Zhao Chang, Dong Xie, Sheng Wang, and Feifei Li. 2022. Towards Practical Oblivious Join. In *Proceedings of the 2022 International Conference on Management of Data.* 803–817.

[21] Hao Chen, Ilaria Chillotti, and Ling Ren. 2019. Onion ring ORAM: efficient constant bandwidth oblivious RAM from (leveled) TFHE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* 345–360.

[22] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates) *(ASIA CCS '17).* Association for Computing Machinery, New York, NY, USA, 7–18. https://doi.org/10.1145/3052973.3053007

[23] Seung Geol Choi, Dana Dachman-Soled, S Dov Gordon, Linsheng Liu, and Arkady Yerukhimovich. 2021. Compressed oblivious encoding for homomorphically encrypted search. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security.* 2277–2291.

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[25] Manuel Costa, Lawrence Esswood, Olga Ohrimenko, Felix Schuster, and Sameer Wagh. 2017. The pyramid scheme: Oblivious RAM for trusted processors. *arXiv preprint arXiv:1712.07882* (2017).

[26] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Paper 2016/086. https://eprint.iacr.org/2016/086 https://eprint.iacr.org/2016/086.

[27] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. 2021. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles.* 655–671.

[28] Ioannis Demertzis, Javad Ghareh Chamani, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2019. Dynamic searchable encryption with small client storage. *Cryptology ePrint Archive* (2019).

[29] Dirk Eddelbuettel. 2022. A Brief Introduction to Redis. arXiv:2203.06559 [stat.CO]

[30] Saba Eskandarian and Matei Zaharia. 2017. Oblidb: Oblivious query processing for secure databases. *arXiv preprint arXiv:1710.00458* (2017).

[31] Francesca Falzon, Evangelia Anna Markatou, Zachary Espiritu, and Roberto Tamassia. 2022. Range Search over Encrypted Multi-Attribute Data. *Proc. VLDB Endow.* 16 (2022), 587–600. https://api.semanticscholar.org/CorpusID:252545892

[32] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, and Srinivas Devadas. 2015. Freecursive ORAM: [Nearly] Free Recursion and Integrity Verification for Position-based Oblivious RAM. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 103–116.

[33] Christopher W. Fletchery, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas Devadas. 2014. Suppressing the Oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 213–224. https://doi.org/10.1109/HPCA.2014.6835932

[34] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. 2016. TWORAM: Efficient oblivious RAM in two rounds with applications to searchable encryption. In *Annual International Cryptology Conference*. Springer, 563–592.

[35] Craig Gentry, Kenny A Goldman, Shai Halevi, Charanjit Julta, Mariana Raykova, and Daniel Wichs. 2013. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies: 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings 13*. Springer, 1–18.

[36] Javad Ghareh Chamani, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2018. New constructions for forward and backward private symmetric searchable encryption. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1038–1055.

[37] Oded Goldreich and Rafail Ostrovsky. 1996. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)* 43, 3 (1996), 431–473.

[38] Michael T Goodrich. 2011. Randomized shellsort: A simple data-oblivious sorting algorithm. *Journal of the ACM (JACM)* 58, 6 (2011), 1–26.

[39] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2011. Oblivious RAM simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop* (Chicago, Illinois, USA) *(CCSW '11)*. Association for Computing Machinery, New York, NY, USA, 95–100. https://doi.org/10.1145/2046660.2046680

[40] S Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. 2012. Secure two-party computation in sublinear (amortized) time. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 513–524.

[41] S. Dov Gordon, Jonathan Katz, and Xiao Wang. 2018. Simple and Efficient Two-Server ORAM. In *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III (Lecture Notes in Computer Science)*, Thomas Peyrin and Steven D. Galbraith (Eds.), Vol. 11274. Springer, 141–157. https://doi.org/10.1007/978-3-030-03332-3_6

[42] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. 2020. Pancake: Frequency smoothing for encrypted data stores. In *29th USENIX Security Symposium (USENIX Security 20)*. 2451–2468.

[43] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 217–233. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss

[44] Thang Hoang, Ceyhun D Ozkaptan, Gabriel Hackebeil, and Attila Altay Yavuz. 2018. Efficient oblivious data structures for database services on the cloud. *IEEE Transactions on Cloud Computing* 9, 2 (2018), 598–609.

[45] Thang Hoang, Ceyhun D. Ozkaptan, Gabriel Hackebeil, and Attila Altay Yavuz. 2021. Efficient Oblivious Data Structures for Database Services on the Cloud. *IEEE Transactions on Cloud Computing* 9, 2 (2021), 598–609. https://doi.org/10.1109/TCC.2018.2879104

[46] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila Altay Yavuz. 2018. Hardware-Supported ORAM in Effect: Practical Oblivious Search and Update on Very Large Dataset. *Proceedings on Privacy Enhancing Technologies* 2019 (2018), 172 – 191. https://api.semanticscholar.org/CorpusID:4007767

[47] T-H Hubert Chan and Elaine Shi. 2017. Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs. In *Theory of Cryptography Conference*. Springer, 72–107.

[48] Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. 2019. Lower bounds for oblivious data structures. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2439–2447.

[49] Stanislaw Jarecki and Boyang Wei. 2018. 3PC ORAM with Low Latency, Low Bandwidth, and Fast Batch Retrieval. Cryptology ePrint Archive, Paper 2018/347. https://eprint.iacr.org/2018/347

[50] Jonathan Katz and Yehuda Lindell. 2014. Introduction to Modern Cryptography. *(No Title)* (2014).

[51] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'neill. 2016. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1329–1340.

[52] Marcel Keller and Peter Scholl. 2014. Efficient, oblivious data structures for MPC. In *Advances in Cryptology–ASIACRYPT 2014: 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, ROC, December 7-11, 2014, Proceedings, Part II 20*. Springer, 506–525.

[53] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient oblivious database joins. *arXiv preprint arXiv:2003.09481* (2020).

[54] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. 2012. On the (in) security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 143–156.

[55] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. 2018. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 297–314.

[56] Kasper Green Larsen and Jesper Buus Nielsen. 2018. Yes, there is an oblivious RAM lower bound!. In *Annual International Cryptology Conference*. Springer, 523–542.

[57] Mingyu Li, Jinhao Zhu, Tianxu Zhang, Cheng Tan, Yubin Xia, Sebastian Angel, and Haibo Chen. 2021. Bringing Decentralized Search to Decentralized Services. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 331–347. https://www.usenix.org/conference/osdi21/presentation/li

[58] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 359–376.

[59] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 279–296.

[60] Muhammad Naveed. 2015. The Fallacy of Composition of Oblivious RAM and Searchable Encryption. Cryptology ePrint Archive, Paper 2015/668. https://eprint.iacr.org/2015/668

[61] Rafail Ostrovsky and Victor Shoup. 1996. Private Information Storage. Cryptology ePrint Archive, Paper 1996/005. https://eprint.iacr.org/1996/005 https://eprint.iacr.org/1996/005.

[62] Simon Oya and Florian Kerschbaum. 2021. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *30th USENIX security symposium (USENIX Security 21)*. 127–142.

[63] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. 2018. PanORAMa: Oblivious RAM with logarithmic overhead. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 871–882.

[64] Benny Pinkas and Tzachy Reinman. 2010. Oblivious RAM revisited. In *Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30*. Springer, 502–519.

[65] Martin Raab and Angelika Steger. 1998. "Balls into Bins" - A Simple and Tight Analysis. In *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM '98)*. Springer-Verlag, Berlin, Heidelberg, 159–170.

[66] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. 2015. Constants count: Practical improvements to oblivious {RAM}. In *24th USENIX Security Symposium (USENIX Security 15)*. 415–430.

[67] Daniel S Roche, Adam Aviv, and Seung Geol Choi. 2016. A practical oblivious map data structure with secure deletion and history independence. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 178–197.

[68] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. 2016. TaoStore: Overcoming Asynchronicity in Oblivious Data Storage. In *2016 IEEE Symposium on Security and Privacy (SP)*. 198–217. https://doi.org/10.1109/SP.2016.20

[69] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. 2017. ZeroTrace: Oblivious memory primitives from Intel SGX. *Cryptology ePrint Archive* (2017).

[70] Sajin Sasy and Olga Ohrimenko. 2019. Oblivious sampling algorithms for private data analysis. *Advances in Neural Information Processing Systems* 32 (2019).

[71] Elaine Shi. 2020. Path oblivious heap: Optimal and practical oblivious priority queue. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 842–858.

[72] Elaine Shi, T. H. Hubert Chan, Emil Stefanov, and Mingfei Li. 2011. Oblivious RAM with O((logN)3) Worst-Case Cost. In *Advances in Cryptology – ASIACRYPT 2011*, Dong Hoon Lee and Xiaoyun Wang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 197–214.

[73] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2018. Path ORAM: an extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)* 65, 4 (2018), 1–26.

[74] Emil Stefanov, Elaine Shi, and Dawn Song. 2011. Towards practical oblivious RAM. *arXiv preprint arXiv:1106.3652* (2011).

[75] Afonso Tinoco, Sixiang Gao, and Elaine Shi. 2022. Enigmap : External-Memory Oblivious Map for Secure Enclaves. Cryptology ePrint Archive, Paper 2022/1083. https://eprint.iacr.org/2022/1083 https://eprint.iacr.org/2022/1083.

[76] Shruti Tople, Yaoqi Jia, and P. Saxena. 2018. PRO-ORAM: Constant Latency Read-Only Oblivious RAM. *IACR Cryptol. ePrint Arch.* 2018 (2018), 220. https://api.semanticscholar.org/CorpusID:4009879

[77] Christopher J. Van Wyk and Jeffrey Scott Vitter. 1986. The complexity of hashing with lazy deletion. *Algorithmica* 1, 1–4 (jan 1986), 17–29. https://doi.org/10.1007/BF01840434

[78] Xiao Wang, Hubert Chan, and Elaine Shi. 2015. Circuit oram: On tightness of the goldreich-ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 850–861.

[79] Xiao Shaun Wang, Kartik Nayak, Chang Liu, TH Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious data structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 215–226.

[80] Peter Williams and Radu Sion. 2012. Single round access privacy on outsourced storage. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 293–304.

[81] Zhiqiang Wu and Rui Li. 2023. OBI: a multi-path oblivious RAM for forward-and-backward-secure searchable encryption.. In *NDSS*.

[82] Leqian Zheng, Zheng Zhang, Wentao Dong, Yao Zhang, Ye Wu, and Cong Wang. 2024. H_2O_2RAM: A High-Performance Hierarchical Doubly Oblivious RAM. arXiv:2409.07167 [cs.CR] https://arxiv.org/abs/2409.07167

[83] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 283–298.

[84] Wenting Zheng, Ryan Deng, Weikeng Chen, Raluca Ada Popa, Aurojit Panda, and Ion Stoica. 2021. Cerebro: A platform for {Multi-Party} cryptographic collaborative learning. In *30th USENIX Security Symposium (USENIX Security 21)*. 2723–2740.

A DAORAM ANALYSIS

In this section, we present the correctness and security proof of DAORAM, analyze its stash size and efficiency, and highlight the concrete improvements.

A.1 Correctness and Security

Correctness. We first prove DAORAM satisfy the correctness defined in Definition 4.1 as below:

THEOREM A.1. *Given the KV store $\mathcal{DB} := \{(k_i, v_i)\}_{i=0}^{n-1}$, where keys are consecutive integers, and a sequence of access operations $\vec{y} = \{(op_i, ek_i, ev_i)\}_{i=0}^{m-1}$ on the KV store, suppose C separately*

(1) processes $\mathcal{DB}$ using DAORAM *where the initialization takes n as input, $2^\gamma > X$, and $m \leq 2^{\alpha+1}(2^\gamma - 1)$ to get $\mathcal{DB}_0$;*
(2) processes $\mathcal{DB}$ just in clear to get $\mathcal{DB}_1$.

Then $\mathcal{DB}_0$ and $\mathcal{DB}_1$ should preserve identical KV pairs with a probability of 1.

Before the proofs, we present two claims implied by the design of recursive ORAMs:

- *Recursive proof.* DAORAM consists of $O(\log n/\log X)$ ORAMs for recursion (cf. Section 2): one is Data ORAM and the others are PosMap ORAMs. Suppose we can prove the Data ORAM can process operations correctly when all PosMap ORAMs guarantee correctness. In that case, we can recursively apply the proof to prove that each PosMap ORAM can process operations correctly because the recursion process is identically repeated except the ORAM size is gradually decreased during the recursion.
- *Limited operations.* The operations on the ORAM consist of only reading and updating the value component in KV pairs (resp. read and write). After the initialization, there are n KV pairs stored in the ORAM no matter if they own values, e.g., a KV pair without value is stored like $(k_i, \perp)$. Such limited operations are common in ORAM since ORAM was originally proposed to simulate RAM where there is a fixed number of entries to be read and written in memory.

Then we can prove DAORAM indeed guarantee the correctness of the Data ORAM on these operations to prove Theorem A.1. For ease of presentation, we use the version of DAORAM without optimizations on reducing groups. The optimized one is too complex to understand but is equivalent to the unoptimized one *in concept* as it only elegantly compresses the zeros in storage.

PROOF. We prove Data ORAM processes each operation correctly by induction. After the initialization of DAORAM, the largest PosMap stores the index of KV pairs in the Data ORAM with PRF inputs: both GC^rs and GC^ps are 0, IC^ps are also set as 0 while IC^r are initialized as 1. Importantly, the following two properties are guaranteed:

- Each pair (k_i, v_i) is **never duplicated and is exclusively placed** encrypted in a block of Data ORAM.
- The block of (k_i, v_i) is either stored in the stash or on the path of Data ORAM **calculated by the PRF** based on the corresponding block in the (largest) PosMap ORAM. To be specific, the KV pair in this block has the key $\lfloor k_i/X \rfloor$.

First, we analyze the two phases in query processing towards the first operation (op_0, ek_0, ev_0). For the query phase, we assume C can get the block with the key of $\lfloor k_i/X \rfloor$ in the largest PosMap ORAM, then the above two properties guarantee it can correctly *remove* the KV pair with key ek_0 from the Data ORAM and read/write it as required. After that, C assigns

$$\mathsf{IC}^r_{j_0} := \mathsf{IC}^r_{j_0} + 1 = 2$$

where $\mathsf{IC}^r_{j_0}$ denotes the PRF individual count used to find ek_0. Here we suppose the reset is not triggered because we guarantee $2^\gamma > X$. As X has to be no smaller than 2 for recursion and the maximal value set for $\mathsf{IC}^r_{j_0}$ is $2^\gamma - 1$, $\mathsf{IC}^r_{j_0}$ must be at least 3 to trigger reset. Thus, C can directly *place* this (possibly updated) pair back to the ORAM correctly according to the incremented $\mathsf{IC}^r_{j_0}$. The removal and placement promise this pair is always the latest and there are no stale copies in the ORAM. For the reset phase, it just evicts a random path as no reset is triggered. The eviction does not affect the two properties because it just pushes KV pairs towards leaves in the ORAM instead of changing their paths assigned. In this way, we can conclude the first operation is completed correctly and the two properties are still preserved.

Second, suppose the first η operations can be done correctly and the two properties still hold, now we prove the $(\eta + 1)$th operation $(op_{\eta+1}, ek_{\eta+1}, ev_{\eta+1})$ can be done correctly when $\eta+1 \leq 2^\alpha(2^\gamma - 1)$. Similarly, the two properties guarantee that C still can correctly remove the pair with key $ek_{\eta+1}$ from the ORAM to read/write it, now we need to analyze the placement, reset, and if the two properties will be broken. There are several cases for $\mathsf{IC}^r_{j_{\eta+1}}$ and the block with key $\lfloor ek_{\eta+1}/X \rfloor$ in the largest PosMap ORAM to be discussed:

(1) When $\mathsf{IC}^r_{j_{\eta+1}} < 2^\gamma - 1$ and $\mathsf{GC}^r = \mathsf{GC}^b = 0$, there is never reset happened on this block and the $(\eta+1)$th operation also does not trigger reset, then this operation is done identically to the first operation: the (possibly updated) pair is stored according to the PRF calculation based on incrementing $\mathsf{IC}^r_{j_{\eta+1}} = \mathsf{IC}^r_{j_{\eta+1}} + 1$, the reset phase only evicts a random path, the two properties still preserves.

(2) When $\mathsf{IC}^r_{j_{\eta+1}} = 2^\gamma - 1$ and $\mathsf{GC}^r = \mathsf{GC}^b = 0$, there is no prior reset on this block but the current operation triggers the reset. C swaps G^r and G^p and sets

$$\mathsf{IC}^p_{j_{\eta+1}} = 0, \mathsf{IC}^r_{j_{\eta+1}} = 1, \mathrm{b} = 1 - \mathrm{b}.$$

Then the (possibly updated) pair is placed by the new $\mathsf{IC}^r_{j_{\eta+1}}$ (under the PRF key K^b). There is still no copy and stale data for this pair in the ORAM. The remarkable change is that after this reset, the other KV pairs tracked by this block can be found correctly only by using IC^ps instead of IC^rs. For the reset phase here, it evicts a random path and does not affect the two properties and correctness. Finally, the two properties still hold but the calculation of PRF is somehow different.

(3) When $\mathsf{IC}^r_{j_{\eta+1}} < 2^\gamma - 1$ and $\mathsf{GC}^r > 0$, there is at least one prior reset on this block but the current operation does not trigger a reset. Then there are two cases:

- $\mathsf{IC}^r_{j_{\eta+1}} = 0$ implies this pair is tracked by $\mathsf{IC}^p_{j_{\eta+1}}$. So C will use $\mathsf{IC}^p_{j_{\eta+1}}$ for removal and then completes partial reset via

setting:

$$\mathsf{IC}^p_{j_{\eta+1}} = 0, \mathsf{IC}^r_{j_{\eta+1}} = 1.$$

- $\mathsf{IC}^r_{j_{\eta+1}} > 0$ implies this individual counter has been reset well. So C is going to remove the pair directly with $\mathsf{IC}^r_{j_{\eta+1}}$ and increment this individual counter.

In both two cases, C can remove the KV pair correctly from the Data ORAM, then C uses the updated $\mathsf{IC}^r_{j_{\eta+1}}$ to place this (possibly updated) pair back. So this pair is tracked well, the two still properties hold. When we consider the reset phase during this operation, recall it picks j such $\mathsf{IC}^r_j = 0$ to access, then it is identical to the above process but with a different key, affecting nothing on the two properties and tracking the selected pair. If all IC^rs are non-zero, the reset phase only evicts a random path, also leaving no effect.

(4) When $\mathsf{IC}^r_{j_{\eta+1}} = 2^\gamma - 1$ and $\mathsf{GC}^r > 0$, there is at least one prior reset on this block and the current operation triggers the reset. This can be done identically to case (2) if all IC^ps are zero and all IC^rs track KV pairs correctly, **which are guaranteed by the two properties and** $2^\gamma > X$. Note that after one reset, one block can complete at least $2^\gamma - 2$ operations before one of IC^rs within it reaches the maximal value to trigger the next reset. That implies case (3) happens at least $2^\gamma - 2$ times and the reset phase picks zero to reset for *at least* $2^\gamma - 2$ times. Between two resets, there is at least one pair accessed, leaving *at most* $X - 1$ pairs whose IC^r are zero, waiting for the reset phase in case (3) to change the values. So once it holds that

$$2^\gamma - 2 \geq X - 1$$

the conditions can be satisfied. Then C can process this case identically to case (2) and preserve the two properties.

Finally, we conclude that the $(\gamma + 1)$th operation is completed correctly and still preserves the two properties, which also concludes our proof. Additionally, we require $m \leq 2^{\alpha+\gamma}$ because we do not allow GC to be repeated, it is independent of correctness but is necessary for security. □

Security analysis. For the security, we first give the following lemma and then present a formal theorem to prove the security of DAORAM.

Lemma A.2. *Given the number of operations denoted by m, there is no repeated input to PRF in* DAORAM *if* $m \leq 2^{\alpha+1}(2^\gamma - 1)$.

Proof. We recall there are four parameters as inputs to PRF (either $\mathsf{PRF}_{\mathsf{K}^0}$ or $\mathsf{PRF}_{\mathsf{K}^1}$) in DAORAM and we prove there must be a distinction between these inputs to calls of the PRF:

(1) Level implies which ORAM is accessed as there are $O(\log_X n)$ ORAMs in DAORAM for *recursion*. To access the ith ORAM, the client uses i as input for PRF calls in this ORAM. It guarantees that any PRF calls on different ORAMs have no repeated inputs.
(2) Key indicates the key of the accessed pair in the current ORAM (level), e.g., for the ith subORAM (level), the key of a pair can be a value from $\{0, 1, ..., 2^{i-1} - 1\}$. It guarantees that any PRF calls on different pairs within one ORAM have no repeated inputs.
(3) GC and IC are two parameters for the same pair. The mechanism of DAORAM promises that GC||IC strictly increases with access to the same pair: 1) each time C retrieves this pair, IC is incremented; 2) when IC reaches the maximal value or it is reset, IC is set to 1 but GC is incremented.

In this way, as the client calculates PRF like

$$\mathsf{PRF}(\mathsf{level}||\mathsf{key}||\mathsf{GC}||\mathsf{IC})$$

where each parameter is set to a fixed length, there is no repeated inputs for PRF calls when GC does not reach its maximal value, i.e., $m \leq 2^{\alpha+\gamma}$. The 2^γ results from each GC is incremented only one of IC within its block reaches the maximal value $2^\gamma - 1$, requiring at least $2^\gamma - 1$ operations to increment IC from 1 to $2^\gamma - 1$. Recall there are two PRF functions alternately applied, so the maximal allowed operations can be multiplied by 2, concluding $m \leq 2^{\alpha+1}(2^{\gamma-1})$. □

Theorem A.3. *DAORAM satisfies the security defined in Definition 4.1, i.e., given two operation sequences $\vec{y_0}$ and $\vec{y_1}$ with the same length and their access patterns in DAORAM denoted by $A(\vec{y_0})$ and $A(\vec{y_1})$, then $A(\vec{y_0})$ and $A(\vec{y_1})$ are computationally indistinguishable, i.e., $A(\vec{y_0}) \equiv_c A(\vec{y_1})$.*

Proof. We adopt a series of transitions to formally prove the security of DAORAM step by step. The basic security guarantee we base on follows the security proof of Path ORAM [73]. For ease of presentation, within this proof, we call the $O(\log n / \log X)$ ORAMs in DAORAM as its subORAMs. WLOG, we assume the keys of KV pairs stored by DAORAM are $\{0, 1, ..., n-1\}$.

(1) **Simulate one subORAM in** DAORAM **with truly random numbers.**
we first establish a simple non-recursive Path ORAM which stores KV pairs $\{(k_i, v_i)\}_{i=0}^{n-1}$ where keys are consecutive integers from 0 to $n-1$. We use it to simulate the Data subORAM in DAORAM for security analysis. Recall DAORAM consists of $O(\log n / \log X)$ subORAMs, we will show how to simulate all of them for security analysis in step (3). For the current non-recursive Path ORAM, we require:
   - C stores a position map for assigning *truly random* paths to each KV pair.
   - C simulates the access in DAORAM. That means, for each integer set $S := \{a, a+1, ..., a+X-1\}$ where $a \in \{X, 2X, ..., (\frac{n}{X} - 1)X\}$[3], C preserves the same parameters and data structure as DAORAM **in local**, e.g., GC and ICs. Then C increments and resets them identically to DAORAM but now C accesses pairs according to the truly random paths instead of PRF outputs.
   For query processing, when C follows a real query to access one pair with a key from S, it also selects another key from S to access its pair for reset or performs a dummy access to pretend reset, which depends on the values of GC and ICs.

   Therefore, given a sequence of operations $\vec{y_b} = \{op_i^b, ek_i^b, ev_i^b\}_{i=0}^{m-1}$, its access patterns can be concluded as

$$\{(\mathcal{P}(ek_0^b), \mathcal{P}(ed_0^b)), ..., (\mathcal{P}(ek_{N-1}^b), \mathcal{P}(ed_{N-1}^b))\}$$

   where ed_i^d denotes the key of the pair accessed for reset, $\mathcal{P}(ek_i^b)$ and $\mathcal{P}(ed_i^b)$ denote the access pattern of retrieving the path assigned to ek_i^b and ed_i^b, respectively. While $\mathcal{P}(ek_i^b)$ corresponds to the access of real query, $\mathcal{P}(ed_i^b)$ is used for simulating the

[3] WLOG, we assume $n \mod X = 0$. Generally, n can be enlarged by adding at most $X - 1$ to satisfy this condition.

reset process. It is either a dummy path retrieve (when $ed_i^b = \perp$) or the retrieve for another item ed_i^b selected by the reset strategy in DAORAM.

(2) **Transition from statistical indistinguishability**
Denote the access patterns via executing $(\vec{y_0}, \vec{y_1})$ on the ORAM above as $(A_1(\vec{y_0}), A_1(\vec{y_1})))$, we conclude $A_1(\vec{y_0}) \equiv A_1(\vec{y_1})$. This is because we always assign a new truly independent random path to a pair after it is accessed. In detail, we follow Path ORAM to analyze this:
- $\forall i \in [m]$, $(\mathcal{P}(ek_i^b), \mathcal{P}(ed_i^b))$ selects two independent random paths as the reset strategy guarantees $ek_i^b \neq ed_i^b$.
- $\forall i, j \in [m]$ and $i \neq j$, $(\mathcal{P}(ek_i^b), \mathcal{P}(ed_i^b))$ and $(\mathcal{P}(ek_j^b), \mathcal{P}(ed_j^b))$ are statistically independent no matter if there are repeated keys in (ek_i^b, ed_i^b) and (ek_j^b, ed_j^b). For distinct keys, they are clearly assigned with independent random paths. For a repeated key, it is assigned with a completely new independent random path after each access.

In total, given any access patterns
$$A_1(\vec{y}) := \{(pt_0^0, pt_0^1), (pt_1^0, pt_1^1)..., (pt_{N-1}^0, pt_{N-1}^1)\}$$
where (pt_i^0, pt_i^1) implies the access patterns of retrieving two paths from the ORAM, it holds
$$\forall b \in \{0, 1\}, \Pr[A_1(\vec{y_b}) = A_1(\vec{y})] = (\frac{1}{n^3} + \frac{2}{n(n-1)})^m.$$
In each retrieve towards two paths, there is no distinction between the order of them since we retrieve them in parallel, e.g., (p_0^0, p_0^1) is identical to (p_0^1, p_0^0). Now we can conclude $A_1(\vec{y_0}) \equiv A_1(\vec{y_1})$.

(3) **Establish recursive ORAM with truly random numbers.**
We repeat step (1) and (2): for each subORAM in DAORAM, we use a non-recursive Path ORAM to simulate it. This means, suppose there are t subORAMs in DAORAM where $t \sim O(\log n / \log X)$, then it holds that:
- C stores *the same content* as DAORAM in the t non-recursive ORAMs. This means the KV pairs stored in each non-recursive ORAM are the same as those stored in the simulated subORAM of DAORAM.
- In each non-recursive ORAM, C performs queries and reset strategy identically to the simulated subORAM in DAORAM. When selecting a pair from a group of pairs for reset, they also select the same pair, which can be done by always picking up the pair with the smallest key. Then the correctness guarantees that they always store the same KV pairs.
- C stores a position map for each non-recursive ORAM to assign and record the truly independent random numbers to KV pairs. It accesses the KV pairs based on the position maps.

So denote the access patterns via executing $(\vec{y_0}, \vec{y_1})$ in the t non-recursive ORAMs as $(A_2(\vec{y_0}), A_2(\vec{y_1})))$, we can define them as
$$A_2(\vec{y_b}) := (A_1(\vec{y_b})^1, A_1(\vec{y_b})^2, ..., A_1(\vec{y_b})^t)$$
where $A_1(\vec{y_b})^i$ indicates the access patterns on the non-recursive ORAM simulating the ith subORAM in DAORAM via executing $\vec{y_i}$. It is easy to see that $A_1(\vec{y_i})^t$ is the $A_1(\vec{y_i})$ in step (2).

(4) **Transition from statistical indistinguishability on recursive ORAM.**
Now we first prove that $A_1(\vec{y_b})^i$ and $A_1(\vec{y_b})^j$ are statistically independent when $i \neq j$ and then prove $A_2(\vec{y_0}) \equiv A_2(\vec{y_1})$.
  (a) $\forall i, j \in [t]$ and $i \neq j$, $A_1(\vec{y_b})^i$ and $A_1(\vec{y_b})^j$ are statistically independent because C always separately and independently assigns truly random numbers to pairs in different non-recursive ORAMs.
  (b) Now with independence between $A_1(\vec{y_b})^i$ and $A_1(\vec{y_b})^j$, we can conclude that given any access patterns
  $$A_2(\vec{y}) := \{A_1(\vec{y})^1, A_1(\vec{y})^2, ..., A_1(\vec{y})^t\}$$
  where $A_1(\vec{y})^i$ denotes access patterns of retrieving random paths on the ith ORAM, it holds that
  $$\Pr[A_2(\vec{y_b}) = A_2(\vec{y})] = \prod_{i=1}^{t} \Pr[A_1(\vec{y_b})^i = A_1(\vec{y})^i].$$
  Note step (2) can be repeated to prove that $\Pr[A_1(\vec{y_0})^i = A_1(\vec{y})^i] = \Pr[A_1(\vec{y_1})^i = A_1(\vec{y})^i]$, so we get $\Pr[A_2(\vec{y_0}) = A_2(\vec{y})] = \Pr[A_2(\vec{y_1}) = A_2(\vec{y})]$, i.e., $A_2(\vec{y_0}) \equiv A_2(\vec{y_1})$.

(5) **Replace truly random numbers with PRF.**
This step is simple: we remove the position map in step (3). Let C replace assigning truly random numbers with PRF calculation which assigns paths identically to that in DAORAM. In this way, we actually construct DAORAM. Please refer to Lemma 1 which guarantees there are no repeated inputs in PRF assigning paths to each item.

(6) **Transition to computationally indistinguishability.**
Denote the access patterns of executing $(\vec{y_0}, \vec{y_1})$ on DAORAM as $(A_3(\vec{y_0}), A_3(\vec{y_1}))$, then it holds that $A_3(\vec{y_0})$ and $A_3(\vec{y_1})$ are computationally indistinguishable, i.e., $A_3(\vec{y_0}) \equiv_c A_3(\vec{y_1})$. This is because $A_3(\vec{y_b})$ and $A_2(\vec{y_b})$ should be computationally indistinguishable, otherwise, an adversary can distinguish PRF and truly random number generator by using them in the ORAMs above. The adversary separately adopts the numbers generated by them to execute $\vec{y_b}$ on the ORAM above. If the access pattern belongs to $A_3(\vec{y_b})$, then the adversary knows the numbers generated are from PRF. Contrarily, it knows they are from the truly random number generator. Therefore, we give the following induction:
$$A_3(\vec{y_0}) \equiv A_2(\vec{y_0}) \equiv A_2(\vec{y_1}) \equiv A_3(\vec{y_1})$$
which concludes our proof.

□

A.2 ORAM Discussion

A.2.1 Stash analysis. Besides the improvement on de-amortization and query costs, DAORAM also outperforms the prior ORAM protocols [32, 47] on client-side stash size. *As the client's storage is valuable and can be limited in reality, this property makes* DAORAM *further more practical than the prior protocols.* This property can be nicely explained for two reasons.

The first reason is the *dummy access* in DAORAM. The fixed protocol [47] is expected to perform $2X$ path retrieval per X accesses, and each retrieval indeed changes the path of one pair in ORAM, possibly incurring stash size increase. DAORAM is much better in this aspect, while it also retrieves $2X$ paths per X real accesses, some of the paths are retrieved as dummy access: such access does not push pairs into the stash, moreover, it evicts both pairs in the

stash and retrieved paths, reducing the stash size. The proportion of dummy access depends on the specific scenario, but it can be at most 50% in some cases, e.g., linear scan on the database.

The other reason is more subtle and is validated by our experiments. During query processing, DAORAM retrieves two paths: one is for the query phase and the other is for the reset phase. Retrieving two paths together allows more aggressive evictions: some blocks in the one path can be evicted deeper in the other path without caching them in the stash, e.g., if $\mathcal{C}$ downloads the two paths and evicts them one by one, then possibly after the target block being assigned with a new path, it cannot be stored in the first path and thus is placed in the stash temporarily, increasing the stash size although it can be evicted into the second path. On the contrary, DAORAM downloads the two paths and evicts them in parallel, so this target block will be directly evicted into the second path without being cached in the stash. This finding can be a potential and general technique to reduce the stash size in $\mathcal{C}$ when executing existing ORAM schemes [32, 47, 73] under client/server paradigm. Differently, such technique is inherent by the de-amortization in our DAORAM while it requires additional modifications and efficiency sacrifice of other ORAM protocols when applying this technique on them.

A.2.2 ORAM comparison. Here we further discuss DAORAM and existing tree-based ORAMs to emphasize its contribution towards more advanced ORAMs. First of all, there are three general components in Path ORAM:

- **The recursion** is used to reduce the size of the position map stored in $\mathcal{C}$, enabling sublinear client-side storage.
- **The path retrieval** determines how $\mathcal{C}$ retrieves the required pair from a path.
- **The eviction procedure** means how $\mathcal{C}$ selects a path and evicts blocks in the path and stash, reducing the client-side stash.

While most existing tree-based ORAMs [21, 34, 66, 78] improve the performance by optimizing the latter two components according to *different scenarios*, DAORAM presents unique and important contributions as it follows [32, 47] to achieve *the best improvement of recursion* up to now, which can work in multiple scenarios especially under the client/server setting. Remarkably, the recursion enhancement is relatively independent of the other two components, allowing us to integrate DAORAM with existing different tree-based ORAMs to propose more advanced ORAMs under different applications. Here we study the integration with two typical and well-known optimized tree-based ORAMs: *circuit ORAM* and *ring ORAM*.

Circuit ORAM [78] is proposed for enhanced performance under *multi-party computation* (MPC) [40]. It achieves the eviction procedure with only linear scans instead of sorting in the famous Path ORAM [73], which makes it very friendly to MPC and widely used in this scenario [49, 58, 84]. Moreover, circuit ORAM is also popular in *trusted execution environments* (TEE) [25, 57, 69] since its eviction consumes only $O(1)$ CPU cache [47] instead of super-logarithmic in Path ORAM. In other words, $\mathcal{C}$ in circuit ORAM only costs $O(1)$ storage. Integrating DAORAM indeed can benefit the circuit ORAM especially when implementing in TEE. This is because DAORAM reduces the number of pairs to be accessed and operated in TEE while preserving the simple eviction and constant CPU cache of circuit ORAM. The effect of DAORAM is still an open problem: the number of operated pairs is still reduced, but the PRF implementations can highly increase the query costs under MPC. We leave it as an open problem as this work does not aim at MPC. Besides, we remark under the typical client/server setting, if $\mathcal{C}$ deploys circuit ORAM to pursue the constant storage, DAORAM can greatly accelerate the query processing.

Ring ORAM [66] addresses the huge communication bandwidth of query processing on ORAM, it is the first to achieve the independence between bandwidth and bucket size in tree-based ORAMs. Specifically, it proposes a new novel path retrieval strategy and also corresponding effective eviction mechanism under the new retrieval strategies. However, compared with Path ORAM, it adds more interaction rounds between $\mathcal{C}$ and $\mathcal{S}$, making it comparable under WAN in time usage only when the block size in ORAMs is large enough. Ring ORAM also selects TEE where interaction round is much cheaper to show its speedup in time usage. Fortunately, as DAORAM generally reduces the number of total KV pairs, integrating it into ring ORAM can gain new performance improvement even under a small block size, further enhancing the practicality of ring ORAM under both WAN and TEE.

There are also lots of other works [13, 14, 41, 67, 68, 76, 81] about tree-based ORAM optimizations which enhance the practicality under some other specific applications in production. For example, rORAM [13] applies the *locality* to improve ORAM on performing range queries. vORAM [67] extends ORAM by enabling variable blocks within ORAM. And recently, a line of works [14, 68, 76] tried to equip ORAM with parallelism under different backgrounds. We remark the recursion optimization in DAORAM is feasible for all of them when they implement recursion to reduce client-side storage.

A.3 Additional ORAM Experiments

A.3.1 Stash size. To validate the effectiveness of *dummy access* and *parallel access* on reducing stash size in DAORAM, besides the prior two ORAM protocols Freeset and Probset, we also separately run DAORAM with

- dummy access and sequential access (denoted by s-Fixset).
- dummy access and parallel access (denoted by p-Fixset).

In this way, s-Fixset is affected by only the dummy access while p-Fixset relies on both dummy and parallel access. To guarantee fairness, we also pick up four typical query distributions to test the stash size. The results are shown in Figure 8. It is shown the size of s-Fixset is always larger than that of s-Fixset but still impressively smaller than Freeset and Probset, demonstrating that the positive impact of parallel access and dummy access.

A.3.2 Obliviousness. We verify the obliviousness of DAORAM by conducting it with 2^{10} queires under six typical query distributions with different skewness. We also run Freeset and Probset to show the consistent dominance of DAORAM. The experimental results are shown in Figure 9. The time usage of Probset is a little unstable, which is consistent with our analysis of the fixed protocol [47]: it performs unstable due to the probabilistic reset operations. Even though we run plenty of queries, its time usage still fluctuates. On the contrary, Fixset is always relatively stable, and performs mostly efficiently.

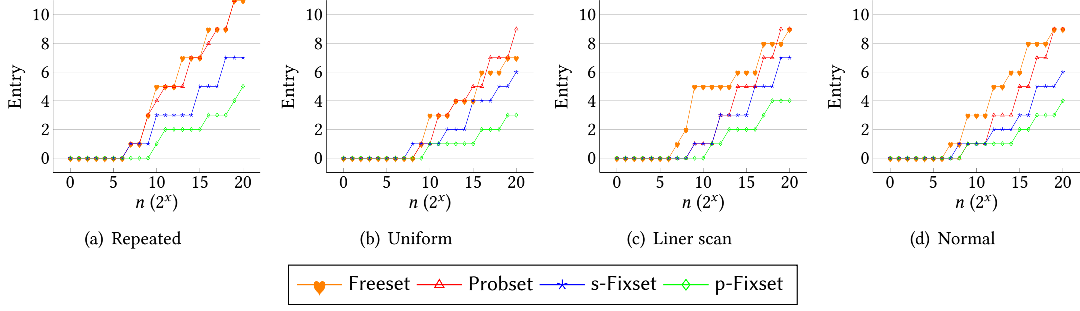


Figure 8: Maximal stash size under different query numbers and distributions. s-Fixset and p-Fixset denote that retrieving two paths in DAORAM sequentially and in parallel, respectively.

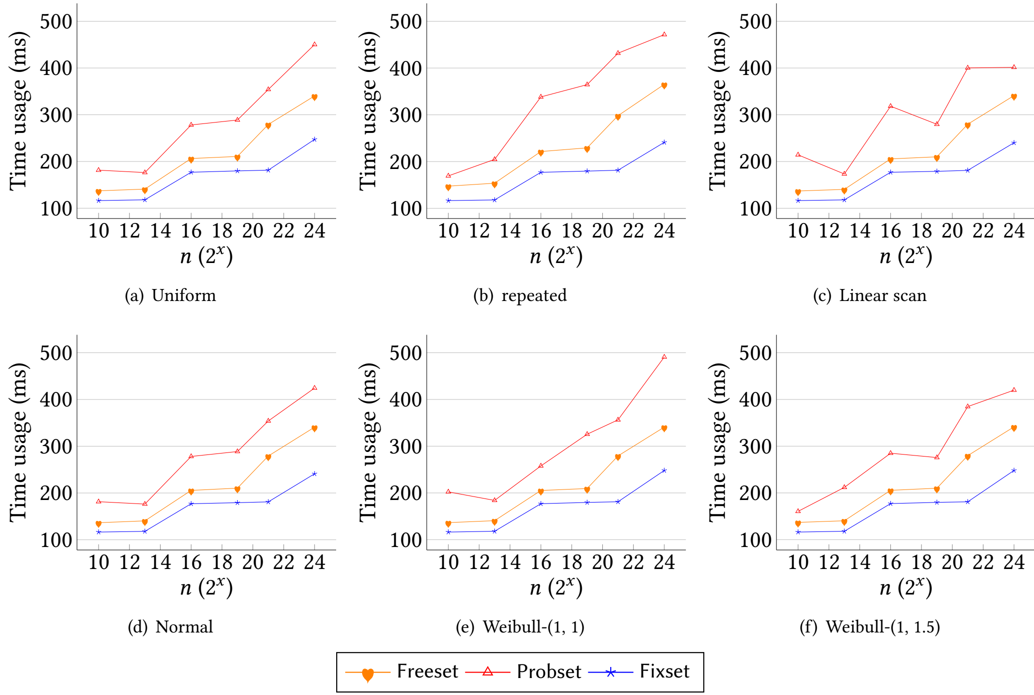


Figure 9: ORAM performance under different query distributions.

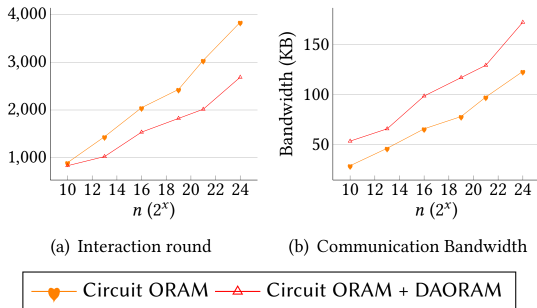


Figure 10: Integration of Circuit ORAM and DAORAM

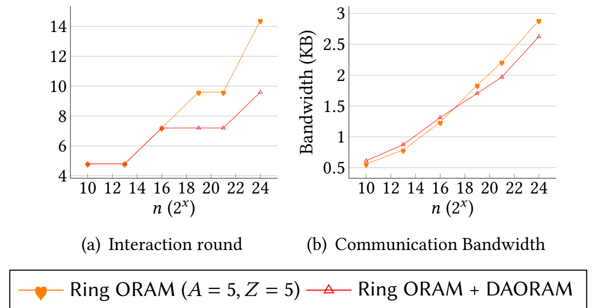


Figure 11: Integration of Ring ORAM and DAORAM

A.3.3 Integration. We integrate DAORAM with circuit ORAM [78] and ring ORAM [66] to show its speedup to existing tree-based ORAMs. That means we run four protocols:

(1) circuit ORAM
(2) circuit ORAM + DAORAM
(3) ring ORAM
(4) ring ORAM + DAORAM

where "+" means the integration. We focus on the theoretical improvement by adapting DAORAM as the underlying position map and demonstrate the potential speedup in Figure 10 and Figure 11.

The figures show that for both circuit ORAM and ring ORAM, the integration reduces the number of interaction rounds required. When n is set as 2^{24}, the integration achieves a 30% reduction in interaction rounds for circuit ORAM and a 33.3% reduction for ring ORAM. As we have observed in the previous experiments, the interaction rounds dominate the total processing time in a WAN setting. Hence the improvement on it can reflect the savings on the total processing time. If these two ORAMs were run locally, the reduction of interaction rounds would be less meaningful and the bandwidth becomes more important. Although DAORAM is at a slight disadvantage due to the need to acquire two paths per access, its higher compression ratio compensates for this constant overhead as the data size n increases. In conclusion, the speedup brought by integrating DAORAM is always meaningful under a general client/server setting even if the client is played by TEE.

B GROUP OMAP ANALYSIS

Correctness and Security. We describe the ODS for the n search trees under our framework as group OMAP, but it also can be imagined as a *large* tree whose root node has n children and each child corresponds to the root node of a search tree among the n search trees. In this way, the correctness and security naturally follow from prior works [67, 79] that design ODS for search trees. The challenge is that we want to access this large tree with as low interaction rounds as possible. The prior works [35, 72] (e.g., Lemma 1 in [35]) have shown that here each group has at most $O(\lambda)$ pairs, and thus the height of the tree is $O(\log \lambda)$. Here we adopt the latest work [27] to further reduce the bound for group size. Following Theorem 6.1, we list the value of $f(n, \lambda)$ when $\lambda = 128$ in Table 6 to show the bound can be very small and practical, e.g., $f(n, \lambda) = 52$ even if $n = 2^{25}$. It is shown $f(n, \lambda)$ increases little with the increase of n, making it especially friendly to **very large databases**. That is one reason why our OMAPs can outperform prior OMAPs significantly better than prior OMAPs as n increases. The interaction round of accessing the group OMAP is nearly unchanged within a wide range of n, and $f(n, \lambda)$ is always no more than $5 \cdot \log n$ when $2^{10} \leq n \leq 2^{25}$ under $\lambda = 128$. Actually, even when $n = 2^{64}$, $f(n, \lambda)$ is only 62 under $\lambda = 128$, which is smaller than $\log n$.

Discussion. We discuss if we can remove the group OMAP to utilize only DAORAM to achieve OMAP as the group OMAP introduces $O(\log \lambda)$ interaction rounds. The answer is negative. Wang et al. [79] points out that applying the hash scheme in [4], an oblivious hash table can be constructed based on an ORAM. This oblivious hash table can be used as an OMAP where each operation is done via three accesses to the ORAM. However, this hash scheme has a probability of $1/p(n)$ failing to complete one operation within only three accesses, and such failure makes it unknown if such an OMAP can be computationally secure. Besides, with the low $f(n, x)$ in practice, the interaction rounds of our OMAPs can be smaller than an OMAP based on this hash scheme as $f(n, \lambda)$ changes slowly with n changed. Overall, our OMAPs are still the most practical OMAP protocols up to now as far as we know.

n	$\log n$	$f(n, \lambda)$	n	$\log n$	$f(n, \lambda)$
2^{10}	10	48	2^{18}	18	50
2^{11}	11	49	2^{19}	19	51
2^{12}	12	49	2^{20}	20	51
2^{13}	13	49	2^{21}	21	51
2^{14}	14	49	2^{22}	22	51
2^{15}	15	50	2^{23}	23	52
2^{16}	16	50	2^{24}	24	52
2^{17}	17	50	2^{25}	25	52

Table 6: $f(n, \lambda)$ when $\lambda = 128$.

C OMAP ANALYSIS

C.1 OMAP Correctness

We do not formally prove the correctness of our OMAPs because it is naturally inherited from the correctness of DAORAM and group OMAP. Imitating the correctness introduced in Appendix B, the group OMAP actually preserves a large tree that has n subtrees and each subtree corresponds to a group under our framework. DAORAM preserves all the information about the root node of this large tree as n KV pairs. Therefore, if DAORAM is correct and the group OMAP (i.e., the ODS for the large tree) is correct with a probability of $1 - \mathsf{negl}(\lambda)$, our OMAPs are also correct with a probability of $1 - \mathsf{negl}(\lambda)$. The challenge is to prove the correctness of DAORAM, which has been formally proven in Appendix A.1.

C.2 OMAP Security

We summarize operations in OMAP as write and read in Section 2. Given the server state st_C and client state $\mathsf{st}_\mathcal{S}$ (after initialization), now we describe them in detail for a further thorough discussion.

- **Read:** $\mathsf{Read}(\mathsf{st}_C, \mathsf{st}_\mathcal{S}, k, v) \rightarrow (\mathsf{st}'_C, \mathsf{st}'_\mathcal{S}, v')$. On input the states $(\mathsf{st}_C, \mathsf{st}_\mathcal{S})$ and a pair (k, v) where $v = \perp$, C and $\mathcal{S}$ interact with each other to run this subroutine, and produce the updated states $(\mathsf{st}'_C, \mathsf{st}'_\mathcal{S})$. If k exists in $\mathsf{st}_\mathcal{S}$, v' is set to the value stored in $\mathsf{st}_\mathcal{S}$ corresponding to k. Otherwise, $v' = \perp$
- **Write:** $\mathsf{Write}(\mathsf{st}_C, \mathsf{st}_\mathcal{S}, k, v) \rightarrow (\mathsf{st}'_C, \mathsf{st}'_\mathcal{S}, v')$. On input the states $(\mathsf{st}_C, \mathsf{st}_\mathcal{S})$ and a pair (k, v) where $v \neq \perp$, C and $\mathcal{S}$ interact with each other to run this subroutine, and produce the updated states $(\mathsf{st}'_C, \mathsf{st}'_\mathcal{S})$. If k exists in $\mathsf{st}_\mathcal{S}$, v' is set to the value stored in $\mathsf{st}_\mathcal{S}$ corresponding to k and the corresponding value in $\mathsf{st}_\mathcal{S}$ is set to v. Otherwise, C inserts the pair (k, v) in $\mathsf{st}_\mathcal{S}$ and $v' = \perp$. *This operation covers both insertion and updating the value component of KV pairs in KV stores.*

The above two operations do not consider deletion and updating the key component of KV pairs. And initially, OMAPs require the two operations should be indistinguishable [30, 79] for the untrusted server $\mathcal{S}$, which makes read operation expensive. Some works [15, 30] present that the query types can be allowed to reveal

for more efficient read, i.e., $\mathcal{S}$ can know if C is reading or writing. For example, in [15], $3 \cdot 1.44 \log n$ interaction rounds are needed for write but only $1.44 \log n$ interaction rounds are needed for read (named FIND in [15]). This is because C does not have to padding the access sequences of read to that of write for hiding the query type. In this paper, we thus divide OMAPs into two kinds: *type-hiding* OMAP and *type-revealing* OMAP to indicate this difference.

Then for the security definition in Section 4.1, we require for any two operation pairs (op_i^0, ek_i^0, ev_i^0) and (op_i^1, ek_i^1, ev_i^1), they should belong to the same operation type (either read or write) if the OMAP protocol is allowed to be type-revealing. And thus we propose the following theorem for all our OMAPs.

THEOREM C.1. *Our construction combining* DAORAM *in Section 5 and any OMAP in Section 6 is type-hiding (type-revealing) OMAP if the OMAP based on is type-hiding (type-revealing)*

PROOF. We first identify the security requirement. There are two components in all our constructions: one is for the data-independent tree using DAORAM and we denote the corresponding access patterns as $(A^0(\vec{y_0}), A^0(\vec{y_1}))$ when executing the sequences $(\vec{y_0}, \vec{y_1})$. The other is for the group OMAPs using an existing OMAP with some modifications, the corresponding access patterns are denoted by $A^1(\vec{y_0})$ and $A^1(\vec{y_1})$ for executing $(\vec{y_0}, \vec{y_1})$. Now we need to prove

$$(A^0(\vec{y_0}), A^1(\vec{y_0})) \equiv_c (A^0(\vec{y_1}), A^1(\vec{y_1})).$$

We formally prove this with the following three parts

(1) DAORAM **security:** $A^0(\vec{y_0}) \equiv_c A^0(\vec{y_1})$.
This directly comes from the security of DAORAM. To execute $\vec{y_b}$, the client picks m (not necessarily distinct) items from $\{(k_1, v_i)\}_{i=1}^n$ to access where k_i are consecutive integers from 0 to $n-1$. Therefore, we apply the security guarantee of DAORAM, which we have demonstrated in Section 5.2 (cf. Theorem A.3).

(2) **Group OMAP security:** $A^1(\vec{y_0}) \equiv A^1(\vec{y_1})$ **with probability** $1 - \mathsf{negl}(\lambda)$**.**
The type-hiding OMAP and type-revealing OMAP differ only in this component. For type-hiding OMAP, $\vec{y_0}$ and $\vec{y_1}$ are required to have only the same length, i.e., the same number of operations. Each operation just accesses the same number of independent random paths. However, type-revealing OMAP needs different numbers of paths to retrieve for different operations. Therefore, when considering type-revealing OMAP, $\vec{y_0}$ and $\vec{y_1}$ are also required to have the same kind of operation in the same position. We still note each operation only accesses a fixed number of independent and random paths.
Then we can discuss two cases to analyze the group ORAM security.
   - **Case 1.** In the first case, there is overflow which implies a group consisting of pairs over the upper bound set by C. Please remark Definition 4.1 requires that the maximal number of total pairs stored in the OMAP should be always no larger than the estimated maximal item number n in the initialization. Then as we claimed in [35, 72] (e.g., Lemma 1 in [35]), C can set a bound of $O(\lambda)$ such that overflow happens with a probability negligible in λ. When this case happens, C stops this protocol, the OMAP fails, incurring the correctness error.
   - **Case 2.** The second case is the main case where each group has at most $w \log n$ pairs. Then no matter which existing OMAP we adopt (like oblivious AVL tree, B tree, and B+ tree), the access patterns of executing $\vec{y_b}$ in the group ORAM can always be defined as

$$A^1(\vec{y_b}) := \{\mathcal{P}_0^b, \mathcal{P}_1^b, .., \mathcal{P}_{m-1}^b\}$$

   where $\mathcal{P}_i$ represents retrieving a fixed number of (not necessarily distinct) random paths from the group ORAM for executing the ith operation in $\vec{y_b}$. We remark 1) the fixed number is deterministically determined by $w \log n$ and the existing OMAP we adopt; 2) the paths retrieved are truly random because the client assigns each pair a truly random number for each access.
   Besides, $\mathcal{P}_i^b$ and $\mathcal{P}_j^b$ are statistically independent because each pair is always assigned a truly independent random number for each access. They are independent no matter if they have overlaps in pairs accessed. For example, even $\mathcal{P}_i^b$ and $\mathcal{P}_j^b$ are access patterns for retrieving the same pair, they are guaranteed to be independent since they just retrieve the same number of independent random paths. Therefore, as both $A^1(\vec{y_0})$ and $A^1(\vec{y_1})$ are retrieving the same fixed number of independent random paths, they are statistically indistinguishable.

In total, with $A^1(\vec{y_0}) \equiv A^1(\vec{y_1})$ in case 2, we can conclude $A^1(\vec{y_0}) \equiv A^1(\vec{y_1})$ if the OMAP does not fail, which happens with a probability of $1 - \mathsf{negl}(\lambda)$ since case 1 happens with only a probability negligible in λ.

(3) **Independence between** DAORAM **and group OMAP**.
This independence is more natural. The DAORAM preserves only the truly random numbers C assigned to pairs in the group OMAP. These random numbers are independent of the DAORAM itself. With this independence, we proceed to the following three steps:
   (a) Given the specific access patterns on DAORAM and group OMAP denoted by $(A^0(\vec{y}), A^1(\vec{y}))$ via executing an operation sequence $\vec{y}$ with the same length to $\vec{y_0}$ and $\vec{y_1}$ (if the OMAP is type-revealing, then they should also have the same operation type in the same position), the group OMAP security guarantees that

$$(A^0(\vec{y_0}), A^1(\vec{y_0})) \equiv (A^0(\vec{y_0}), A^1(\vec{y})),$$
$$(A^0(\vec{y_1}), A^1(\vec{y_1})) \equiv (A^0(\vec{y_1}), A^1(\vec{y})),$$

   if the group OMAP does not happen overflow.
   (b) Next, we can use the DAORAM security, if a PPT adversary can distinguish $(A^0(\vec{y_0}), A^1(\vec{y}))$ and $(A^0(\vec{y_1}), A^1(\vec{y}))$, then it can distinguish $A^0(\vec{y_0})$ and $A^0(\vec{y_1})$ by padding them with the same access patterns on group OMAP, i.e., $A^1(\vec{y})$. Recall DAORAM guarantees $A^1(\vec{y_0}) \equiv_c A^1(\vec{y_1})$, so it holds that

$$(A^0(\vec{y_0}), A^1(\vec{y})) \equiv_c (A^0(\vec{y_1}), A^1(\vec{y}))$$

   (c) Replacing $A^1(\vec{y})$ according to independence and the group OMAP security, we conclude

$$(A^0(\vec{y_0}), A^1(\vec{y_0})) \equiv_c (A^0(\vec{y_1}), A^1(\vec{y_1}))$$

   if the OMAP has no overflow on group OMAP.

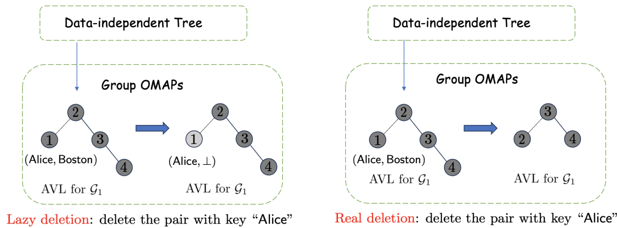


Figure 12: The two deletion strategies for a query.

Finally, to include overflow, we suppose there exists a PPT adversary to distinguish the two access patterns with a probability of 1 in overflow, as it happens with only a probability of $\mathsf{negl}(\lambda)$, we can conclude a PPT adversary can distinguish $(A^0(\vec{y_0}), A^1(\vec{y_0}))$ and $(A^0(\vec{y_1}), A^1(\vec{y_1}))$ with an **advantage** of

$$\mathsf{negl}(\lambda) + (1 - \mathsf{nege}(\lambda)) \cdot \mathsf{negl}(\lambda)$$

which is clearly negligible in λ and thus concludes our proof.

□

Among works considered in this paper, the works [15] are type-revealing OMAP and the baseline OMAP [79] is type-hiding OMAP, our OMAPs can significantly accelerate all of them.

C.3 Delete and Updating Keys

Throughout this work, we do not consider deletion and updating the key component of KV pairs in KV stores because they can be decomposed into the two operations we defined in Appendix C.2, namely read and write with some minor modifications. Following the lazy deletion strategy [77], we can complete the deletion via read: if this pair does not exist in the KV store, then C failed to read it. Otherwise, C additionally sets the value component of this pair as $\perp$ in read, marking it has been deleted. Here C does not adopt write because write can be more expensive than read in type-revealing OMAPs. Updating the key component in KV pairs is more complex, we first lazily delete the old pair and then insert the new pair, i.e., one read and one write operation. In the remainder of this section, we further discuss the two operations in more aspects to understand them in practice, including performing them with real deletion instead of lazy deletion, and their impact on the data structure.

Deletion. The lazy deletion does not affect the data structure under our framework at all as the data structure is *determined by only the key component* of KV pairs. Setting the value component to be $\perp$ only marks one node in the data structure as *invalid* but still preserves it to keep the data structure unchanged. Instead, a real deletion removes one node from the data structure, causing the structure to be different. Recall our framework consists of two oblivious components: *the data-independent tree* and *search trees for groups*. The former is always fixed after the initialization of OMAP and independent of any deletion and insertion, but the search trees can be affected by the real deletion, e.g., if the deletion removes one node from the search tree like AVL for a group, then we have to rotate this tree to keep it balanced. We illustrate this in Figure 12. Such differences also make the two deletion strategies different in performance, in type-revealing OMAPs, the cost of lazy deletion

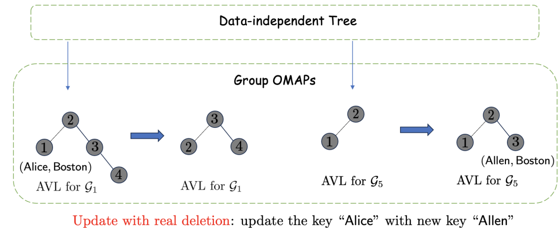


Figure 13: An example of updating key "Alice ⟶ Allen" changes at most two search trees for groups. The node of Alice is removed from $\mathcal{G}_1$ and the node of Allen is inserted into $\mathcal{G}_5$.

is much cheaper than that of real deletion. We remark the real deletion cost is equivalent to that of write because they require the same maximal number of rotations in the search tree [79]. A possible shortage of lazy deletion is that the invalid nodes can result in additional costs on query processing since the invalid nodes potentially make the tree height larger. However, this happens only when there is a large proportion (e.g., 50%) of pairs being lazily deleted.

Updating keys. Updating the key component in KV pairs is the most expensive operation among all operations discussed in this work. This is because it requires both deletion of the old pair and insertion of the new pair. Updating the key component does not affect the data-independent tree but certainly changes the search trees for groups, because of the newly inserted data. Hence an update might change one or two search trees depending on whether the lazy or the real deletion strategy is adopted. We illustrate this with a simple example in Figure 13. Correspondingly, under the two kinds of deletion strategies, the cost of update is equivalent to the combination of one read plus one write and two write operations, respectively. This makes the cost of updating keys with lazy deletion still much cheaper than that of updating keys with real deletion in type-revealing OMAPs. To this end, in type-revealing OMAPs, we recommend using lazy deletion in practice and really deleting items only when the computation and bandwidth resources are sufficient enough.

D ADDITIONAL OMAP EXPERIMENTS

In this section, we show additional experimental results to further evaluate our constructions and compare them with prior works.

D.1 Search, Delete and Update

In Section 7, we evaluated the insertion operation to test write in OMAPs. Here we test more operations of OMAPs including the search (to capture read), lazy delete, real delete, and updating the key component of KV pairs based on the two types of deletions.

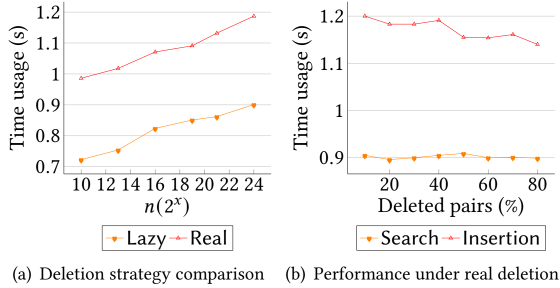
Search. We test the search operations only in type-revealing OMAPs as type-hiding OMAPs execute search identically to insertion to guarantee obliviousness and insertion has been well evaluated in Section 7.2. In type-revealing OMAPs, the cost of search is significantly lower than that of insertion because it does not require structural adjustments and, therefore, avoids the redundant padding accesses needed during insertion. We run 100 search operations on the prior type-revealing OMAPs [15, 30] and our new OMAPs based on them with database size varying from 2^{10} to

n		2^{10}			2^{13}			2^{16}			2^{19}			2^{21}			2^{24}		
Time components		T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3
AVL	prior (s)	0.02	0.98	1.00	0.03	1.24	1.27	0.06	1.56	1.62	0.09	1.84	1.93	0.11	2.04	2.15	0.13	2.31	2.44
	ours (s)	0.04	0.59	0.63	0.05	0.68	0.73	0.06	0.75	0.81	0.08	0.78	0.85	0.08	0.85	0.93	0.09	0.89	0.98
	speedup (%)	-100.0	39.8	37.0	-66.7	45.2	42.5	0.0	51.9	50.0	11.1	57.6	56.0	27.3	58.3	56.7	30.8	61.5	59.8
B+	prior (s)	0.03	1.02	1.05	0.05	1.16	1.21	0.08	1.33	1.41	0.10	1.64	1.74	0.13	1.78	1.91	0.16	1.91	2.07
	ours (s)	0.01	0.69	0.70	0.01	0.71	0.72	0.02	0.78	0.80	0.03	0.79	0.82	0.03	0.80	0.83	0.04	0.82	0.86
	speedup (%)	66.7	32.4	33.3	80.0	38.8	40.5	75.0	41.4	43.3	70.0	51.8	52.9	76.9	52.9	56.5	75.0	57.1	58.5

Table 7: Time usage of search. T_1 is calculation time, T_2 is the communication time, and T_3 is the total processing time.

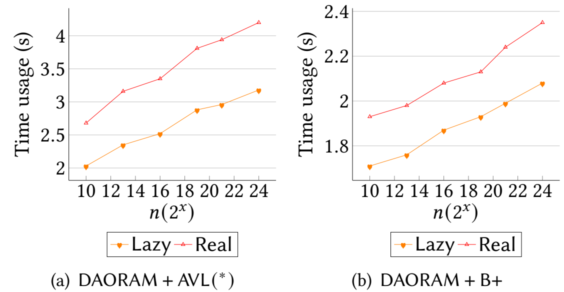
n		2^{10}		2^{13}		2^{16}		2^{19}		2^{21}		2^{24}	
Communicate		round	band (KB)	round	band (KB)	round	band (KB)	round	band (KB)	round	band (KB)	round	band (KB)
AVL	prior (s)	30	115.20	38	184.83	48	294.91	56	401.41	62	492.03	70	627.20
	ours (s)	22	45.57	22	47.10	24	49.92	24	52.22	24	53.76	26	56.32
	speedup (%)	26.7	60.4	42.1	74.5	50.0	83.1	57.1	87.0	61.3	89.1	62.9	91.0
B+	prior (s)	28	50.18	36	82.94	44	123.90	48	147.46	56	200.70	64	262.14
	ours (s)	20	20.48	20	22.02	22	24.83	22	27.14	22	28.67	24	31.23
	speedup (%)	28.6	59.2	44.4	73.5	50.0	80.0	54.2	81.6	60.7	85.7	62.5	88.1

Table 8: Interaction rounds and communication bandwidth of search.



(a) Deletion strategy comparison (b) Performance under real deletion

Figure 14: Performance of DAORAM+B+ under deletion.



(a) DAORAM + AVL(*) (b) DAORAM + B+

Figure 15: Performance of OMAPs for updating the key component in KV pairs with different deletion strategies.

2^{24}. The processing time and communication metrics are recorded in Table 7 and Table 8. They show that the speedup on search is smaller than that of insertion, but it can still be as high as 59.8%. Given that OMAP has been extensively studied in the past decade and the search operation has already been optimized much [15, 30], such an improvement to SOTA works can be very valuable. Similar to insertion, in the two tables, communication occupies over 90% time usage in query processing. For the search operation, our OMAPs also mainly reduce the communication time to enhance performance. As our lower complexity allows fewer interaction rounds and less communication bandwidth with n increasing, the improvement is also more impressive under a larger n. Additionally, we note that the client-side algorithms in our OMAPs are more complex, thus when $n = 2^{10}$, our OMAP based on the AVL shows a negative speedup on T_1 although it has a lower bandwidth. As n increases, T_1 can be sped up positively by our OMAPs since the bandwidth is much less, implying a much smaller number of KV pairs to be calculated.

Delete and update. We test the other operations in type-revealing OMAPs, but only in some specific properties as all of them can be decomposed into the read and write. For deletions, we separately evaluate the performance of lazy deletion and real deletion with DAORAM+B+ and depict the results in Figure 14. As expected, lazy deletion is much cheaper than real deletion as they are done by the cheap read and expensive write, respectively. To show the effect of lazy deletion, we provide an empirical evaluation in Figure 15(b). While lazy deletion does not affect query cost, real deletion can potentially reduce the cost of other operations if enough data is deleted, leading to a rebalancing of the underlying search tree and a reduction in its height. This trend is evident in the graph. Since we initially started with exactly 2^{24} data items, we observe that when half of the data is deleted, the query cost decreases for the first time. This demonstrates that lazy deletion is more advantageous when only a small proportion of items are removed. For updating the key component of KV pairs, we compare the update on the four OMAPs with different deletion strategies in Figure 15, which again shows

that the cost of update with lazy deletion is still much cheaper than that of update with real deletion in these OMAPs. And these results show this update is much more expensive than the deletion, i.e., around 2× slower than deletion with the same deletion strategy as this update additionally executes an insertion.

D.2 Obliviousness

To validate the obliviousness of our OMAPs, we evaluate our OMAPs in two aspects: *data obliviousness* and *query obliviousness*. The former implies that the dataset distribution is different while the latter means the distinct query distributions.

Data obliviousness. ORAM always preserves KV stores with consecutive keys, thus the dataset distribution is identical. But OMAP is designed to support dataset with different distributions. For OMAP data obliviousness, we run our constructions on both the real-world dataset PTB and the synthetic dataset. The key length in both of them is 19 bytes as the maximal key in PTB owns 19 bytes. We set them with the same size and access via insertion and search queries. These queries are uniformly distributed and the experimental results are shown in Figure 16. It is shown under the same data size, the processing time on PTB and synthetic dataset is nearly the same. So the performance on the two datasets is nearly identical, validating the obliviousness under different data distributions.

Query obliviousness. For query distributions, in Section 7, we adopted queries under the uniform distribution to evaluate the ORAMs and OMAPs. This follows a series of works [18, 27, 74] about obliviousness and is because the obliviousness property guarantees the query cost is nearly identical under any query distribution (as claimed in [18, 27, 74]). Intuitively, the query processing in the view of $\mathcal{S}$ should be nearly identical to avoid leaking sensitive information. To validate this claim, we run 2^{10} queries generated from 6 distinct and typical distributions on OMAPs and these distributions own different skewness. We test only insertion and search as other operations are decomposed into them. We show the average time usage per query in Figure 17. It indeed shows that the performance of the OMAPs on the same kind of operation is nearly the same under these different query distributions, validating the rationality of evaluating the query cost with only the uniform distribution in the prior experiments.

To explain the obliviousness, the readers may naturally notice that the procedure and algorithms in $\mathcal{S}$ are always designed to perform identically under different access patterns (for obliviousness), e.g., yielding the same interaction rounds, the same communication bandwidth under any query distribution, and the identical query cost. We remark actually the procedure in C can be non-oblivious but incurs only *ignorable* cost variance under the client-server paradigm considered in this work. The client-side algorithms are often required to be oblivious only when implementing them in secure enclaves [26], which is studied by another line of recent works [8, 59, 82].

D.3 Network Conditions

Our work in fact proposes the most practical and efficient *recursive* ORAM and OMAPs up to now. It reduces both the interaction rounds and communication bandwidth to enhance efficiency. However, the two reductions have different impacts on query processing. So here we conduct the four OMAPs compared in Section 7 **locally** to make interaction round nearly free. In this way, we remove the effect of interaction round reduction and can focus on the communication bandwidth reduction. The experimental results are shown in Figure 18. Then we can separately figure out the impact of interaction round reduction and bandwidth reduction.

- The bandwidth reduction has little impact on improving communication time as it occupies at most 3ms while the communication time under WAN costs at least 0.59s in our prior experiments.
- The bandwidth reduction has a more positive impact on calculation time. The bandwidth reduction implies there are fewer pairs retrieved to be calculated. Hence the total time in local is still greatly reduced although the communication time reduction affects it little.

With the above claims, now we can apply Table 4, Table 5, Table 7, and Table 8 to analyze the impacts. The speedup in calculation time T_1 comes from bandwidth reduction while the speedup of communication time T_2 nearly completely results from interaction round reduction as bandwidth reduction has little impact on communication. The readers can easily calculate the different impact of the two reductions on different n, for example, when $n = 2^{24}$, there are around 1.09s and 0.12s time enhancement of search based on B+ tree from interaction rounds and bandwidth reductions, respectively. Please note the proportion differs according to n and OMAPs, but basically, the impact of round reduction is much more important than bandwidth reduction.

In addition to the prior experiments under WAN conditions with a latency of 38ms, we further quantify the speedup under varying latencies. We acquire multiple cloud servers from different cities to represent C under various network conditions. We depict the OMAP insertion and search performance when $n = 2^{24}$ in Table 9 and Table 10, detailing the corresponding latency, time usage, and speedup. These latencies reflect network conditions typical of most real-world applications. The bandwidth is consistently set to 100Mbps. The results align with expectations: as network conditions worsen and latency increases, the improvement in T_1 changes only slightly, since the bandwidth reduction is independent of latency. The interaction rounds take up a larger portion of the total time, leading to a more significant speedup. This makes reducing interaction rounds even more crucial. Therefore, we conclude that our new OMAPs become increasingly interesting in networks with higher latency.

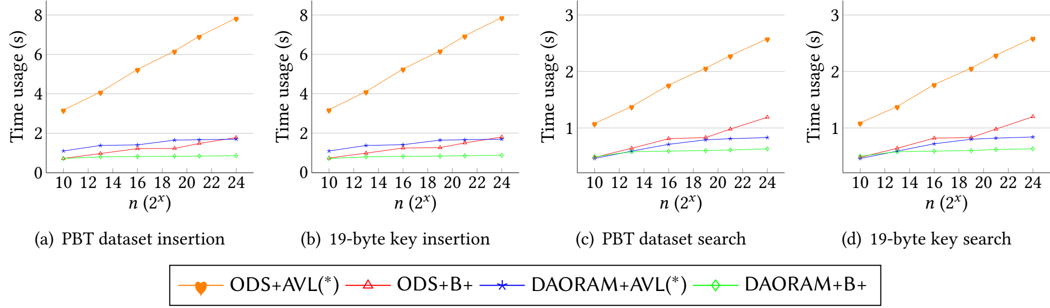


Figure 16: Data obliviousness: processing time on PTB dataset (1KB block).

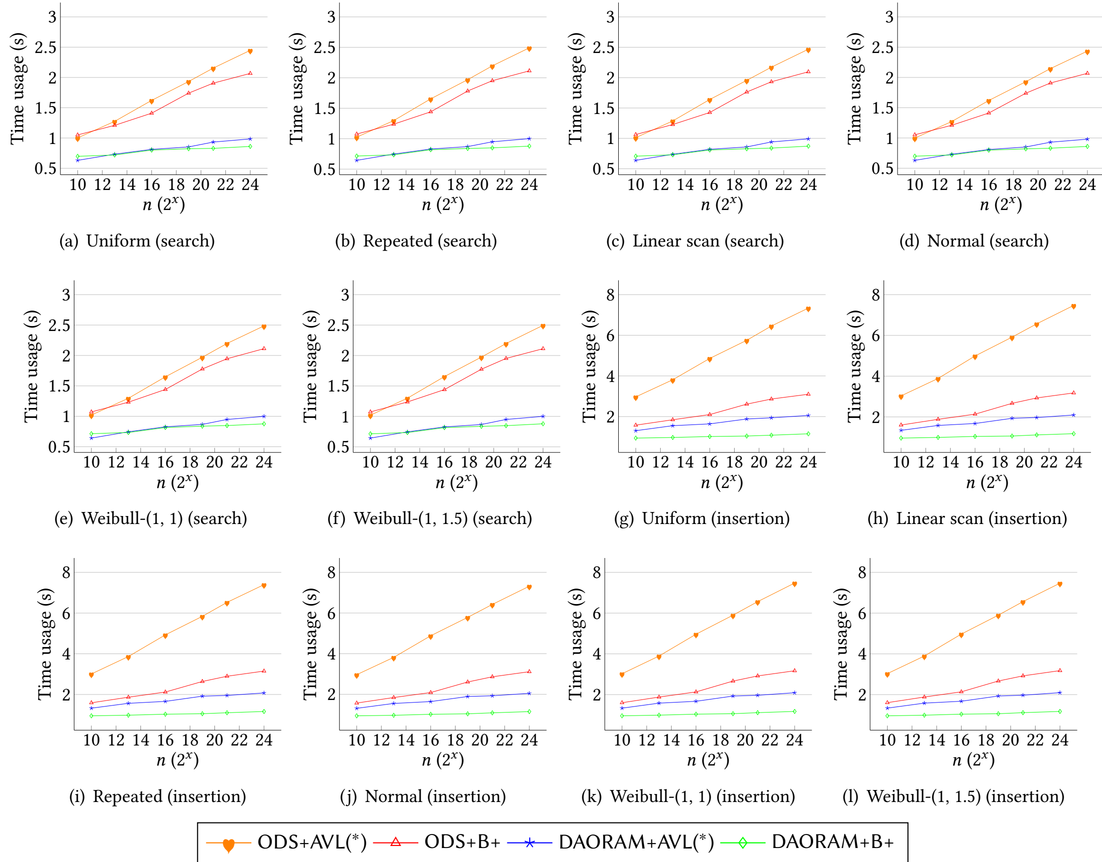
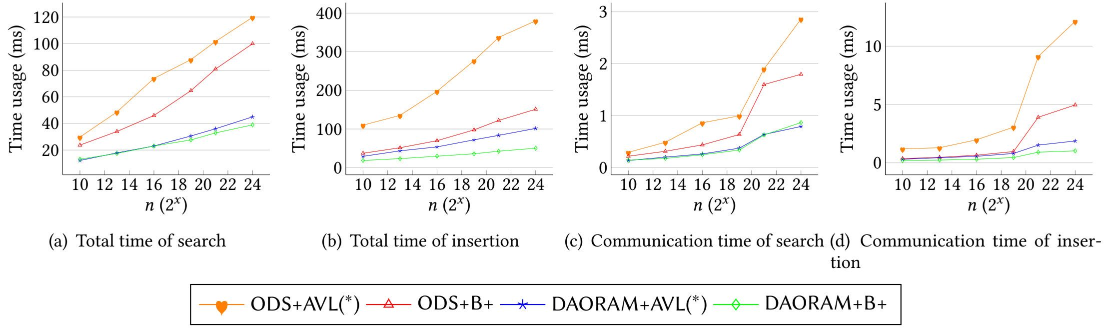


Figure 17: Query obliviousness: OMAP performance under different query distributions.



(a) Total time of search (b) Total time of insertion (c) Communication time of search (d) Communication time of insertion

Figure 18: Query processing in local.

City		Hangzhou (12ms)			Tokyo (35ms)			Kuala Lumpur (82ms)			San Francisco (154ms)			London (208ms)		
Time components		T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3
AVL	prior (s)	0.36	2.16	2.52	0.37	6.54	6.91	0.36	15.79	16.15	0.38	29.20	28.88	0.37	39.19	39.56
	ours (s)	0.11	0.65	0.76	0.11	1.76	1.87	0.12	4.28	4.40	0.11	7.61	7.72	0.11	9.91	10.02
	speedup (%)	69.4	69.9	69.8	70.3	73.1	72.9	66.7	72.9	72.8	71.1	73.9	73.3	70.3	74.7	74.7
B+	prior (s)	0.17	0.92	1.09	0.17	2.69	2.86	0.16	6.38	6.54	0.16	11.49	11.65	0.17	16.23	16.40
	ours (s)	0.06	0.28	0.34	0.06	0.76	0.82	0.06	1.81	1.87	0.06	3.21	3.27	0.06	4.41	4.47
	speedup (%)	64.7	69.6	68.8	64.7	71.7	71.3	62.5	71.6	71.4	62.5	72.1	71.9	64.7	72.8	72.7

Table 9: Time usage of insertion under different latency. T_1 is calculation time, T_2 is the communication time, and T_3 is the total processing time.

City		Hangzhou (12ms)			Tokyo (35ms)			Kuala Lumpur (82ms)			San Francisco (154ms)			London (208ms)		
Time components		T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3
AVL	prior (s)	0.13	0.72	0.85	0.13	2.19	2.32	0.13	5.26	5.39	0.12	9.15	9.27	0.13	13.09	13.22
	ours (s)	0.09	0.28	0.37	0.10	0.83	0.93	0.09	2.01	2.10	0.09	3.52	3.61	0.10	4.87	4.97
	speedup (%)	69.4	69.9	69.8	70.3	73.1	72.9	66.7	72.9	72.8	71.1	73.9	73.3	70.3	74.7	74.7
B+	prior (s)	0.15	0.61	0.76	0.15	1.78	1.83	0.15	4.26	4.41	0.15	8.08	8.23	0.15	10.81	10.96
	ours (s)	0.04	0.30	0.34	0.04	0.74	0.78	0.05	1.72	1.77	0.04	3.17	3.21	0.05	4.15	4.20
	speedup (%)	73.3	50.8	55.3	73.3	58.4	57.4	66.7	59.6	59.9	73.3	60.8	61.0	66.7	61.6	61.7

Table 10: Time usage of search under different latency. T_1 is calculation time, T_2 is the communication time, and T_3 is the total processing time.